POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Master of Science in Computer Engineering

PATTERN-BASED AND VOICE-BASED VIRTUAL ASSISTANTS FOR THE DESIGN OF IFML MODELS

Master Graduation Thesis by:
Davide Molinelli
id. 884889

Advisor:
Marco Brambilla

Academic Year 2017-2018

Success consists of going from failure
to failure without loss of enthusiasm
Winston Churchill

Acknowledgments

Se è vero che la vita è una strada da intraprendere verso mete sconosciute, ringrazio il mio relatore Marco Brambilla per aver accettato di essere la mia guida in questo tratto di percorso così importante per me. Non dimenticherò i consigli, la disponibilità ad ascoltarmi e i modi gentili. Grazie per non avermi voluto indicare la strada, ma averla voluta percorrere insieme a me.

Ringrazio te, Carlo Bernaschina, per avermi fatto appassionare al tuo lavoro e a quella che io amo chiamare "la tua creatura". Adesso la sento anche un po' mia. Grazie per gli infiniti insegnamenti, per l'instancabile pazienza e per la tua umiltà. Sei stato quello che per un bambino è il supereroe della sua infanzia. Il mio modello di ispirazione in ambito accademico e professionale. Quello che ogni giorno mi ricorda che c'é ancora tanto da imparare.

Ringrazio gli amici e colleghi: siete stati i compagni di viaggio che qualsiasi viandante vorrebbe avere al suo fianco lungo il cammino. Il mio ombrello nei giorni di pioggia, la mia coperta nelle notti di freddo. E il sole caldo nei giorni sereni. Mi avete insegnato tanto. Siete entrati nella mia vita e dovete rimanerci.

Ringrazio infine le stelle comete del mio itinerario: i miei genitori. Non ricordo un giorno in cui vi siate spente e non abbiate vissuto per me. Sebbene sia parso molto distratto in questi anni, e con la testa rivolta in tanti pensieri, ho un sensore dentro al petto che sa riconoscere ogni sacrificio fatto per me. Questo lavoro è frutto del figlio che avete cresciuto ed educato. Se questo risultato lo rende orgoglioso, deve rendere orgogliosi anche voi.

Abstract

In an increasingly digital world, which makes available to all people an almost unlimited amount of services through web and mobile platforms, a fundamental role is played by user interfaces (UI). Indeed, user interfaces represent the means by which users can view information and interact with systems. Without a doubt, the success or failure of an application can be determined by its ease of use and the appearance through which it displays itself to the end user. An application that makes intuitive and straightforward the navigation and the content search, the operations, and data retention, is a system that is more likely to win the competition in a market characterized by the presence of numerous copies of similar services. A famous quote in the field of User Experience points out that people ignore designs that ignore people. In other words, it is of primary importance that it is not the user who has to adapt to the characteristics and interaction methods of a new application, but rather the application itself that has to be designed according to the user's needs and habits. In this context, during the whole modeling and development process of a system, the use of validated standard structures, such as those represented by design patterns, allow obtaining final applications that present a high degree of usability.

In this thesis work we will describe the development of support tools for Model Driven Development, which promote the use of design patterns in the generation of models for the rapid prototyping of web and mobile applications. In particular, we will show a tool that allows the generation, customization and automatic insertion of design patterns within a model. Secondly, we will analyze an algorithm able to detect design patterns already present in a model, to facilitate the changes in the model itself. Finally, we will present a second approach to develop models using design patterns, based on the use of an assistant that allows building models by means of voice commands, or to guide the developer in the realization of complete models, automatically generating them at the end of a brief interaction.

Sommario

In un mondo sempre più digitale, che mette a disposizione di tutte le persone una quantità pressoché illimitata di servizi attraverso piattaforme web e mobile, un ruolo fondamentale è svolto dalle interfacce utente. Esse, infatti, costituiscono lo strumento per mezzo del quale gli utenti possono visualizzare le informazioni e interagire con i sistemi. Senza ombra di dubbio si può affermare che il successo o il fallimento di un'applicazione può essere determinato, oltre che dalle funzionalità che offre, anche e soprattutto dalla sua facilità di utilizzo e dall'aspetto con cui essa si presenta all'utente finale. Un'applicazione che riesce a rendere semplice e intuitiva la navigazione e la ricerca di contenuti, l'esecuzione delle operazioni e la conservazione dei dati è un sistema che ha più probabilità di vincere la concorrenza in un mercato caratterizzato dalla presenza di numerose copie di uno stesso servizio. Una citazione popolare nel settore dell'User Experience attesta che le persone ignorano i design che ignorano le persone. In altre parole, è di primaria importanza che non sia l'utente a doversi adattare alle caratteristiche e le modalità di interazione di una nuova applicazione, ma sia l'applicazione stessa a essere progettata secondo le esigenze e le abitudini dell'utente. In questo contesto, durante tutto il processo di modellazione e sviluppo di un sistema, l'utilizzo di strutture standard validate, come quelle rappresentate dai design patterns, consentono di ottenere applicazioni finali che presentano un alto grado di usabilità.

In questo lavoro di tesi descriveremo lo sviluppo di strumenti di supporto al Model Driven Development, che promuovono l'utilizzo dei design pattern nella generazione dei modelli per la prototipazione di applicazioni web e mobile. In particolar modo, mostreremo uno strumento che permette la generazione, la personalizzazione e l'inserimento automatico di design patterns all'interno di un modello. In secondo luogo, analizzeremo un algoritmo in grado di individuare e riconoscere design patterns già presenti all'interno di un modello, per agevolare le operazioni di modifica del modello stesso. Infine, presenteremo un secondo approccio di sviluppo dei modelli tramite design patterns, basato sull'utilizzo di un assistente che permette di costruire modelli per mezzo di comandi vocali, o di guidare lo sviluppatore alla realizzazione di un modello completo, generandolo automaticamente al termine di una breve interazione che consente di apprendere le specifiche dell'applicazione da progettare.

List of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The present chapter introduces the main problem tackled in the thesis work, including the scenario in which it can be framed, and an overview of the designed solution. The chapter ends with a brief description of the structure of the given document.

1.1 Context

The development of a new mobile or web application requires a great effort in the planning of the corresponding abstract model. Indeed, the design of the model represents one of the most delicate steps in the whole process of building and implementing a system: without exaggerating, it could determine the success or failure of the application itself.
A model built through appropriate criteria that take into account the usability and user experience, resulting from the use of similar systems, will be more likely to materialize in an intuitive application, in which the user knows how to navigate, can find what seeks and, consequently, feels gratified.
In this sense, the combined use of regular patterns conceived specifically to ensure order, ease of navigation, modularization, and orientation within the application, represents one of the most effective methods for successful model design.
In the context of the model-driven engineering, it is possible to take advantage of this approach to help the developer in the construction of the models, providing model editors with validated constructs, in which only parameters need to be set, allowing - among other advantages - to improve and speed up the building process in a non-negligible way.

1.2 Problem Statement

At state of the art, the simple implementation of design patterns (made available inside a modeling editor), may not be sufficient to guarantee to the developer their practical, intuitive, and correct use in the model construction process: a model is also and above all the result of the connections and combinations of pure components in an appropriate way that it is not currently possible to be automatically controlled.
The construction of a model is still largely dependent on the imagination and the skill of the developer: it is a non-deterministic process and therefore hard to be managed by a modeling editor that is unable to understand which parts can match which pattern, in order to provide tips and guidelines.

However, the advent of neural networks and artificial intelligence can open new and promising scenarios that have not yet been tested in the field of model-driven engineering. In particular, a smart model editor, equipped with artificial intelligence, may be able to anticipate the development of a model, building a model starting directly from the application specifications, which the editor can interpret. More in details, the specifications can be read and understood through:

- a neural network for natural language processing that interprets a textual document formatted in a proper way.
- a voice assistant that is able to interact with the developer, ask questions and understand the specifics through a simple conversation.

In this way it is also possible to reverse the process of creation and subsequent management of the new model: if previously, the developer constructed a model that had to be interpreted and corrected by the editor, with the use of artificial intelligence it is now possible to leave to the editor the task of constructing a model to which the developer can subsequently make changes. Once again, this approach would allow a significant reduction of errors, a non-negligible acceleration in the construction process and a noticeable reduction of design efforts, without limiting the creativity of the developer, since the standardization of the components of a model, based on patterns, is a concept distinct from that of the development of attractive, unique and original user interfaces that have always been the added value of an application.

1.3 Contributions of the Thesis

The primary purpose of the work is to encourage the use of patterns within the model-driven development, offering rigorous assistance and support criteria during the modeling process. In particular, we intend to achieve the proposed objective by integrating a pattern generator into a modeling editor and implementing a voice assistance system able to support the developer step by step, offering guidance tools and advanced commands, depending on the level of experience of the modeler.

1.4 Document Structure

- Chapter 2 presents the theoretical concepts that define the background of the work.
- Chapter 3 analyzes in detail the related works published by the scientific community.
- Chapter 4 describes the proposed approaches and the designed solution.
- Chapter 5 discusses the experimental results and shows a complete study case.
- Chapter 6 contains the conclusion of the work and exposes possible future evolutions of the system.

Chapter 2

Background

The information presented in this chapter aims to provide the reader with the theoretical concepts used in the thesis work, essential for the comprehension of the following chapters.
At first, the key notion of *pattern* is presented (section 2.1). We will analyze the role that this conceptual element has assumed over time in the field of software engineering and the importance it plays nowadays in software development.
Secondly, in section 2.2, the focus moves on the presentation of Model Driven Development (MDD), a specific branch of software engineering that promotes the uses of models and model transformations as key ingredients of software development. We will see that developers use modeling languages to specify system requirements under different perspectives.
Section 2.3 examines in depth the Interaction Flow Modeling Language (IFML), a domain-specific modeling language used in the field of Model Driven Development to specify and describe a system at a high level of abstraction.
Section 2.4, instead, exposes the IFMLEdit.org framework, which denotes the starting point from which the planned infrastructure has been developed.
The chapter ends with the presentation of the last theoretical pillar of the work: speech recognition technologies in the field of artificial intelligence (section 2.5). We will explore the world of the vocal assistants and, in particular, the main features that characterize Alexa, the software created by the research team of Amazon and used as a model within this thesis work.

2.1 Pattern

An expanse of dunes in the Sahara desert; the characteristics movements of an explorer bee; the famous painting *The Kiss*, by Gustav Klimt; the design of modern urban architecture. All these proposed images are very different from each other and finding commonalities among them represents a task as strange as it is difficult to think. However, although in different contexts and with different shades of meaning, they all share a fundamental conceptual element: pattern.
Starting from an informal and general definition, a pattern represents a draw, a model or a recurrent scheme used to describe the repetition of a certain sequence within a set of raw data, or the regularity observed in a dynamic phenomenon, both in space and time [37]. In this context and referring to the previously presented scenarios, a series of dunes, pyramidal mounds of sand generated and modeled by the wind, represent a recurrent scheme (as well as the primary component) in a desert. In the same way, but with different acceptations, the dance of an explorer bee is the

result of an instinctive and rhythmic gesture around the same space, with the intent to signal the proximity of food to the rest of the swarm. Again, the most famous painting by Gustav Klimt, *The Kiss*, is characterized by repetitive themes and texture motifs, woven into the tunics of the two lovers. Finally, modern urban architecture is characterized by well known and accredited structural elements and design principles. Carefully observing Nature, it is possible to find patterns in almost everything, from inorganic compounds to the organic ones, from microorganism to the macro ones, from plants to animals.

It should not be surprising to know that human beings guide their perception of the world on the very concept of pattern. According to [39], a pattern is, first of all, a mental concept derived from the human experience. The experience and the human reflection lead to recognizing recurring patterns in real objects, that are considered instances of the patterns themselves. As reported in [30], there is an abundant literature on habits and patterns in human behaviour, e.g. [24], [32]. Without going into the interesting debate between Cartesians and pragmatists [32], it is reasonable to state that several theories have been formulated [24] in attempt to explain the formation, use and change of habits and patterns in human behaviour, e.g. the Theory of Reasoned Action [22], the Social Cognitive Theory [7], the Theory of Interpersonal Behavior [41], or the Theory of Planned Behavior [3]. From the learning of these theories, we can deduce that the intelligence of human beings lies not only in recognition of existing patterns, but also and above all in the ability to exploit and reproduce them or create new ones in order to solve new problems arising in new contexts.

The definition, the research and the use of patterns are processes now explored and validated in all humanistic and scientific fields, but the first attempt to catalog patterns dates back to 1977, when the Austrian Christopher Alexander *et al.* published [4], an essay on architecture and city planning, considered even today (after four decades) one of the greatest bestsellers of architecture.

As the authors wrote in the introduction, the 253 patterns collected in the book together form a sort of language able to offer solutions in any specific contexts where it can be applied and free to evolve under the impact of new experiences and observations.

As reported in [39]:

> *"According to Alexander, each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. In this sense, a pattern represents a relationship between forces that keep recurring in a specific context and a configuration which resolves these forces. Besides, a pattern is also a rule that explains how to create a particular configuration which resolves the forces within that context."*
>
> Riehle

It is interesting to note how Alexander's work has significantly influenced many other scientific and engineering fields. In particular, twenty years after his publication, pattern architectures made their way in software engineering thanks to the work of Gamma *et al.* [23]. Gamma affirms that a design pattern encodes good practices guiding developers in solving design problems. Concrete patterns are expressed using classes and objects, but they nevertheless represent solutions to recurrent problems in

particular contexts. This is the most widely spread notion of pattern today, adopted by many other researchers, including Beck & Johnson [10], Schmidt and Coplien [17].

2.1.1 Design Pattern in Software Development

In software engineering, design patterns are considered powerful knowledge-sharing tools because they encapsulate developers' experience and provide a common vocabulary for communication across domains. Design patterns can be part of the entire development cycle, starting from the design, passing through the implementation and finally the maintenance of the software.

> *"The benefits of design patterns during software development have been the subject of many papers. Beck [10] suggests that patterns generate architectures. Similarly, Ram et al. [38] propose the Pattern Oriented Technique for developing systems via patterns as collaborations of design patterns. Also, during the maintenance of systems, software engineers often identify code and design smells and look for refactoring of patterns to remove them. These refactoring and pattern transformations are documented to be reused in the future [29]. Lange and Nakamura [33] demonstrate that pattern can serve as a guide in program exploration and thus make the process of program understanding more efficient. Through a trail of pattern execution, they show that if patterns were recognized at a certain point in the understanding process, they could help in "filling in the blanks" and in further exploring a system, improving thus its understandability. [...] Other papers focus on the selection of design patterns and attempt to draw rules to combine these patterns during development. Guéhéneuc et al. [25] present a recommender system to help software engineers in choosing among the 23 design patterns by Gamma et al."*
>
> Khomh, [30]

2.1.2 Reverse Engineering with Design Patterns

In the particular context of reverse engineering applied to software development, there are at least two important themes that can be mentioned: pattern detection and pattern-based components recovery.
As previously quoted, studies have been conducted to prove that the use of design patterns in software development can drastically improve the comprehension of the system and the code. Consequently, their use has a potentially positive impact on the quality, ordering, and cleaning of the software, reducing the maintenance costs (in terms of time and efforts) [46]. The pattern-based components recovery, instead, is aimed to analyze a system in order to identify the components and the interrelationships that characterized it, create representations of the same system in another form at a higher level of abstraction, and reconstruct parts referable to design patterns through the use of automatic, semi-automatic or manual recovery techniques and a repository of abstract design principles and schemes (as reported in the work by Keller *et al.*).

2.2 Model Driven Development

In all the traditional engineering disciplines, the design of complex systems requires a remarkable effort. In this regard, a verified and consolidated approach over the centuries, to tackle and overcome the problem in question, concerns the definition of models that describe in their entirety the systems to be built, according to the functional requirements that they must satisfy.

> *"No one would imagine constructing an edifice as complex as a bridge or an automobile without first constructing a variety of specialized system models. Models help us understand a complex problem and its potential solutions through abstraction."*
>
> Selic, [40]

This is the point: the use of models allows designers and developers to focus on the key aspects of the system while omitting details of secondary importance [19].
A formal definition of Model Driven Development (MDD) is reported in [26]:

> *"Model Driven Development is a software-engineering approach consisting in the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle."*
>
> Falzone

More in detail, the *abstraction* to which the definition refers indicates the capability to generalize, classify and aggregate elements: models are simplified or partial representation of reality, defined in order to accomplish a task and solve a problem.

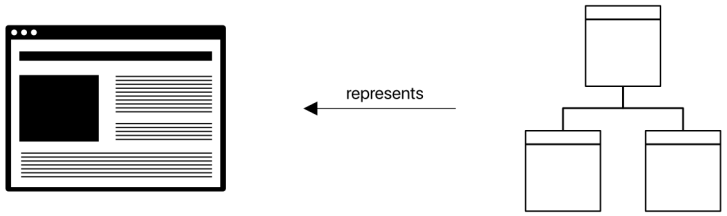


Figure 2.1: Model (on the right) reflects the essential features of the real system (on the left)

Although the main purpose of a model is primarily descriptive, it also has a prescriptive role, as it imposes rules on the definition and evolution of the systems, essentially determined by the infrastructure on which the generation of the model itself is based, other than by its expressive power.
Nowadays, MDD technologies refer to a four-layer infrastructure that represents a hierarchy of model levels where each one represents an instance of the level above, except the top (Figure 2.2). The bottom level (traditionally called M0) holds the

user data, i.e., the real objects that software is designed to manipulate. Level 1 (M1), instead, is said to manage a *model* of the M0 user data. User models we referred until now, reside at this level as they represent an abstraction of the instances of M0. The next level (M2) is commonly named *metamodel* and represents a further abstraction of the main properties of the instances of M1. We can see M2 as a model of the model instance M1 because, in a practical sense, metamodels constitute the definition of a modeling language, since they provide a way of describing the whole class of models that can be represented by that language. In the end, the top level (M3) symbolizes the third level of abstraction, able to model the essential features of an instance of a metamodel. For this reason, level M3 is known as *meta-metamodel*[1].

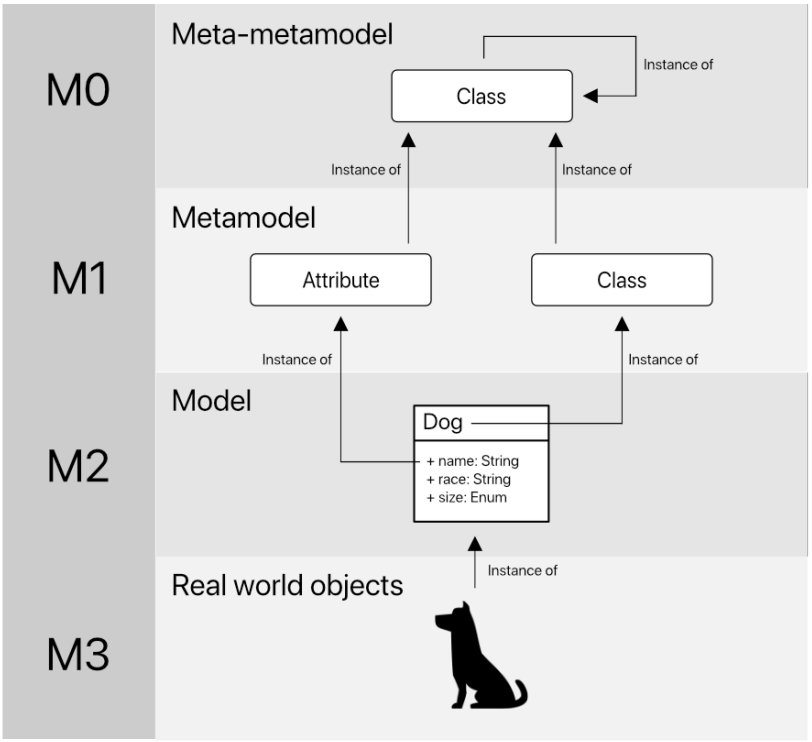


Figure 2.2: Traditional Model Driven Development infrastructure

2.2.1 Modeling Languages

Modeling languages represents one of the main ingredients of Model Driven Development. A modeling language is a tool that lets the designer specify the model of their systems. In other terms, it consents the definition of a concrete representation of a conceptual model through graphical representations, textual specifications, or both [14]. More in detail, two big classes of languages can be identified:

[1]Note that the operation of abstraction of the model could potentially be repeated infinite times. In other words, one could define infinite levels of metamodeling. However, in practice, meta-metamodels represents a level of abstraction that can be used to define itself, so it usually does not make sense go beyond this level of abstraction.

- *Domain Specific Modeling Languages (DSLs)*, such as IFML (Interaction Flow Modeling Language [36]): are languages designed specifically for a certain domain, context or company, in order to ease the task of people that need to describe things within a particular domain. Among the other examples of domain-specific languages are included the well-known HTML markup language for Web page development, VHDL for hardware description languages, Mathematica and MatLab for mathematics and SQL for database access.
- *General-Purpose Modeling Languages (GPLs)* such as UML (Unified Modeling Language [44]): are languages that can be applied to any sector or domain for modeling purpose. State machines and Petri Nets constitute other famous examples of general purpose languages.

2.2.2 Modeling Transformations

The other crucial ingredient of Model Driven Development is symbolized by model transformations. Model transformations can be thought of as rules defined at the metamodel level and applied at the model level [14]. According to a nice comparison reported in [19], model transformations can be seen as functions or programs that take in input models and, depending on the type of the output, can be classified into:

- *Model to Model transformations*: they map source models into target models which can be instances of the same or different metamodels. Figure 2.3 shows an example of Model to Model transformation: a Place Chart Net (PCN) is generated from an IFML model[2].
- *Model to Text transformations*: they focus on the generation of a textual artifact (in particular, code generation) from models. Figure 2.4 shows an example of a Model to Text transformation where an IFML model is mapped into an HTML code block.

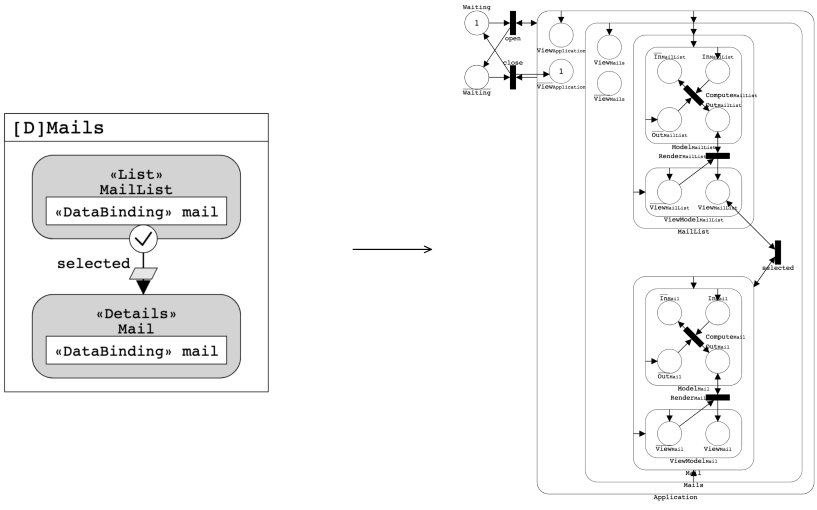


Figure 2.3: Example of Model to Model Transformation

[2]Note that the input model is built using IFML, a *Domain Specific Modeling Language*, while the output model is an instance of PCN, a *General-Purpose Modeling Language*, so the transformation mapped a model from a metamodel to a different one

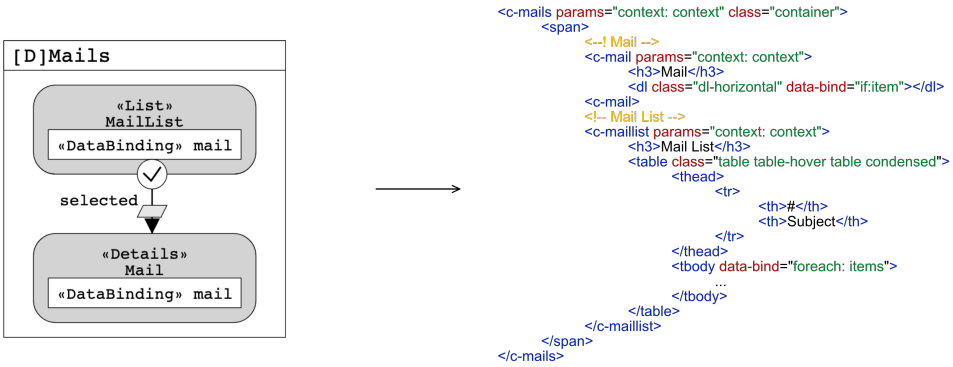


Figure 2.4: Example of Model to Text Transformation

2.2.3 Model Driven Architecture Models

In 2001, the Object Management Group (OMG), a consortium of software vendors and users from industry, government, and academia, announced its *Model Driven Architecture (MDA)* initiative, which offers a conceptual framework for defining a set of standards in support of MDD [40].
The primary goals of MDA are portability, interoperability, and reusability through architectural separation of concerns [42].
MDA defines a special concept of models, that distinguishes those models that take into account the details of the underlying hardware and software (platform) from those that do not [26]. In particular, MDA enumerate three different levels of abstraction (Figure 2.5):

- *Computation-Independent Model (CIM)*: the most abstract modeling level, which represents the context, requirements, and purpose of the solution without any binding to computational implication. It exactly represents what the solution is expected to do, but hides all the IT-related specification, to remain independent of if and how a system will be (or currently is) implemented. [14]

- *Platform-Independent Model*: the modeling level that describes the behavior and structure of the application, regardless of the implementation platform. Notice that the PIM is only for the part of the CIM that will be solved using a software-based solution and that refines it in terms of requirements for a software system. The PIM exhibits a sufficient degree of independence to enable its mapping to one or more concrete implementation platforms. [14]

- *Platform-Specific Model (PSM)*: Even if it is not executed by itself, this model must contain all required information regarding the behavior and structure of an application on a specific platform that developers may use to implement the executable code. [14]

A set of mappings between each level and the subsequent one can be defined through model transformations. Typically, every CIM can map to different PIMs, which in turn can map to different PSMs.

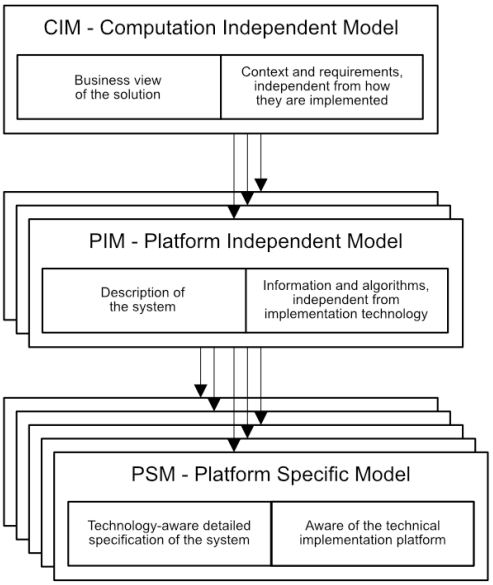


Figure 2.5: The three levels of modeling abstraction codified in MDA

2.3 Interaction Flow Modeling Language

The Interaction Flow Modeling Language (IFML) is a domain-specific modeling language standardized by OMG in 2014.

> *"IFML supports the platform-independent description of graphical user interfaces for applications accessed or deployed on such systems as desktop computers, laptop computers, PDAs, mobile phones, and tablets. The focus of the description is on the structure and behavior of the application as perceived by the end user. The description of the structure and behavior of the business and data components of the application is limited to those aspects that have a direct influence on the user's experience."*
>
> OMG, [36]

.

From the analysis of the definition, it follows that IFML is a modeling language whose generated models lie on the PIM-level of the MDA models subdivision. This brings several benefits, as it raises the abstraction level of the front-end specification (isolating it from implementation-specific issues), improves the coordination work in the development process, permits the explicit representation of the different perspectives of the front end (from the content and organization interface, passing through the interaction and navigation options, up to the connection with the business logic) and enable the communication of interface and interaction design to

nontechnical stakeholders, enabling the early validation of the requirements [15]. In this context, the role of the Interaction Flow Modeling Language is to provide a stable set of concepts that can be used to characterize the essential aspect of the user's interaction, through a software application interface: the provision of stimuli, the capturing and processing of such stimuli by the application logic, and the update of the interface based on such processing.

2.3.1 Main Modeling Concepts

Concerning the popular Model-View-Controller (MVC) architectural pattern (Figure 2.6), typical of the interactive applications, the focus of IFML is on the View component. Furthermore, IFML describes how the View references or depends on the Model and Control parts of the application. More in-depth:

- With respect to the View, IFML deals with the view composition and the description of the elements that it exposes to the user for interaction.
- Concerning the controller, IFML lets the designer specify the effects of user interactions and system events on the application by defining the relevant events that the controller must take care of.
- Concerning the model, IFML allows for the specification of the references to the data objects that represent the state of the application, as well as the reference to the actions that are triggered by the interaction of the user.

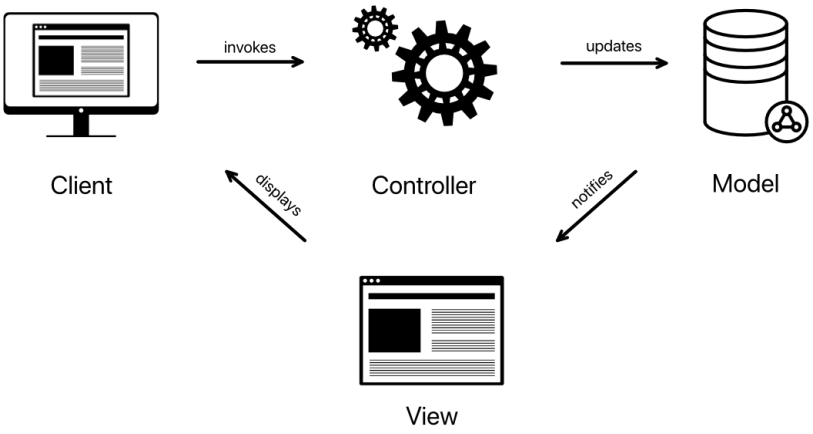


Figure 2.6: The Model-View-Controller architecture of an interactive application

Modeling the user interface and interaction with IFML amounts to addressing a wide range of specifications, in terms of:

- *view structure*: it refers to the composition of the view, i.e., its partition into independent visualization units, which can be displayed simultaneously or in mutual exclusion, and can be nested hierarchically.
- *view content*: it alludes to the set of data elements published from the application to the user and the input data generated by the user towards the application. In other words, this specification carries out the task of providing for the filling of the views with contents.

- *events and transitions*: it concerns the user's interaction within the application, in terms of commands that trigger events and actions that are required by the user. Events and transitions are managed at the level of business logic and can perform changes on the model and consequently on the state of the view.

- *parameter bindings*: it consists on the definition of the input-output dependencies between view components and between view components and actions.

In order to satisfy this requirements, the IFML meta-modeling [11] is composed of the following fundamental elements:

- *ViewElement*: represents the essential classifier for the specification of the application structure and it specializes into *ViewContainer* and *ViewComponent*. As the name suggests, a ViewContainer defines a container into which the interface components are allocated: it supports the visualization of the content and the interaction of the user. Each view container can be internally structured in a hierarchy of sub-containers. In other words, view containers allow the nesting operation. This means that the expressive power of the modeling language permits to build a rich Internet application where the main window can contain multiple tagged frames, which in turn may be characterized by several nested panes. The control on the visibility of the nested view containers works through the definition of a set of parameters that define how the containers must be displayed in the application. For example, according to the constraints imposed by the language, children view containers nested within a parent view container can be displayed simultaneously or in mutual exclusion. A ViewComponent, instead, represents the main content of view containers. They can denote the publication of content (e.g., a list of objects) or the input of data (e.g., entry forms), *et alia*. A view component can have input and output parameters. For example, a ViewComponent that shows the details of an object has an input parameter corresponding to the identifier of the object to display; a data entry form exposes as output parameter the values submitted by the user; a list of items exports as output parameter the item selected by the user.

- *Event*: constitutes the object through which it is possible to trigger events and perform operations. In other words, events support user interaction inside the application. Events can be associated with ViewContainers and ViewComponents in order to submit input forms, select items in a list, move from a visualization page to another, *et alia*. Usually, events are mapped to button elements inside a user interface (technically speaking, this is the case of *user events* and *selection events*), but it is not the rule: they can also refer to operations triggered by the system, in a transparent way respect to the user side (in this case, we talk about *system events*).

- *Action*: it can be described as a black box that refers to an operation fired by a system or user event and executed at the level of the controller (in the MVC architecture pattern). Generally, actions are procedures that cause the updating of the model and consequently can produce modifications also in the state of the user interface (i.e., the view). An action element can receive data in input (that we call parameter and represents the information needed to perform the associated operation) and return results (which can be shown to the user in the form of details, or can serve to fulfill a list with a set of items, *et alia*)

- *Interaction Flow*: a construct that represents the communication element among ViewContainers, ViewComponents, Events, and Actions. The role of this fundamental construct can change, relatively to the elements involved in the connection. In particular, we can distinguish two type of interaction flow:
  - *Navigation Flow*: a flow that can connect an Event to a ViewElement or an Action. As the name suggests, it determines the navigation from an element to another element that could be translated in a change of the visualized content in the user interface (for example, a change of page, assuming that the target element is a ViewContainer), or in the motion of an operation from the client side to the server side (in the case the target element is an Action).
  - *Data Flow*: a flow that can connect a ViewComponent to another ViewComponent in order to exchange data, without producing any interaction.
- *ParameterBinding*: is the element that registers the connection among source and target parameters of an interaction flow. Indeed, as one can guess from the given definition, an interaction flow can transport information in the form of input and output bindings. This means that it is possible to create unidirectional connections between source parameters to corresponding target parameters, transferring in this way the data contained in the source element to the target one.

Figure 2.7 and Table 2.1 respectively show a complete example of an IFML model and the list of the elements which compose the IFML metamodel (including a visual representation of each element and eventually a concrete instance of the same in a real application).

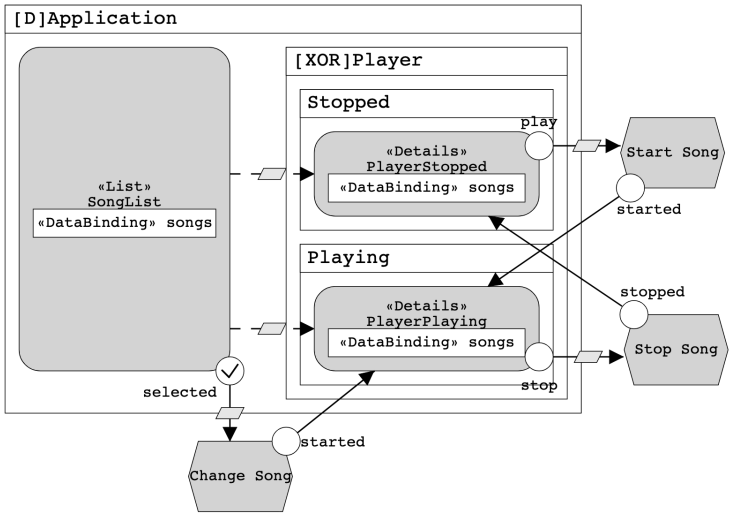


Figure 2.7: Example of a complete IFML model of a media player application

IFML Main Concepts		
Concept	IFML Notation	PSM
ViewContainer		• Web Page • Window • Page
ViewComponent		• HTML List • Input Form
Event		
Action		• Update Database • Send Email
Navigationflow		• Sending and receiving parameters in HTTP request
Dataflow		
ParameterBinding		

Table 2.1: Main IFML concepts and notations

2.4 IFMLEdit.org

IFMLEdit.org is an online framework for the specification of IFML models and the generation of code for web and mobile architectures [12]. The system is realized using web technologies, and it comprises five main components:

1. *Model Editor*: a general-purpose visual editor that consents the developer to insert elements in the model (by means of drag & drop operations), edit their properties, define interactions and express hierarchy relations and information flows (Figure 2.8).
2. *PCN Generator and Simulator*: it allows the developer to generate a PCN from an IFML model, and simulate its behavior (Figure 2.9).
3. *Web Code Generator and Emulator*: it maps an IFML model into a web application, and emulate its operation inside the framework. Moreover, it permits the download of the corresponding source code (Figure 2.10).
4. *Mobile Code Generator and Emulator*: it transforms an IFML model into a cross-platform mobile application and emulates its operation inside the framework. Furthermore, it enables the download of the corresponding source code (Figure 2.11).
5. *Model-to-JSON Transformation Framework*: a generic rule-based transformation framework that, given as input an IFML model, generates its description in a JSON format, letting to export and re-import models.

The traditional workflow, within the framework, is the following: initially, the developer edits the IFML model of an application using the online editor. Subsequently (and optionally), the developer maps the model into a PCN and simulates the network, in order to understand the dynamics of the application in response to events. Next, he moves on the web or mobile emulator, where the model is automatically transformed into the source code of a fast prototype, for a cross-platform mobile language. The emulator executes the code and the programmer tests and validates the prototype. Finally, the developer downloads the code and turns the validated prototype into a real app, by customizing the look&feel and replacing the mock-up data access and skeleton operation API calls with real ones.

The framework just described symbolized the starting point of our work, since it presented all the essential features we searched, where integrate our infrastructure, namely:

- an open-source online platform, with which it was possible to interact through HTTP requests.
- an intuitive and lightweight model editor, oriented to the design of user interfaces, easily understandable and modifiable (from the point of view of the source code).

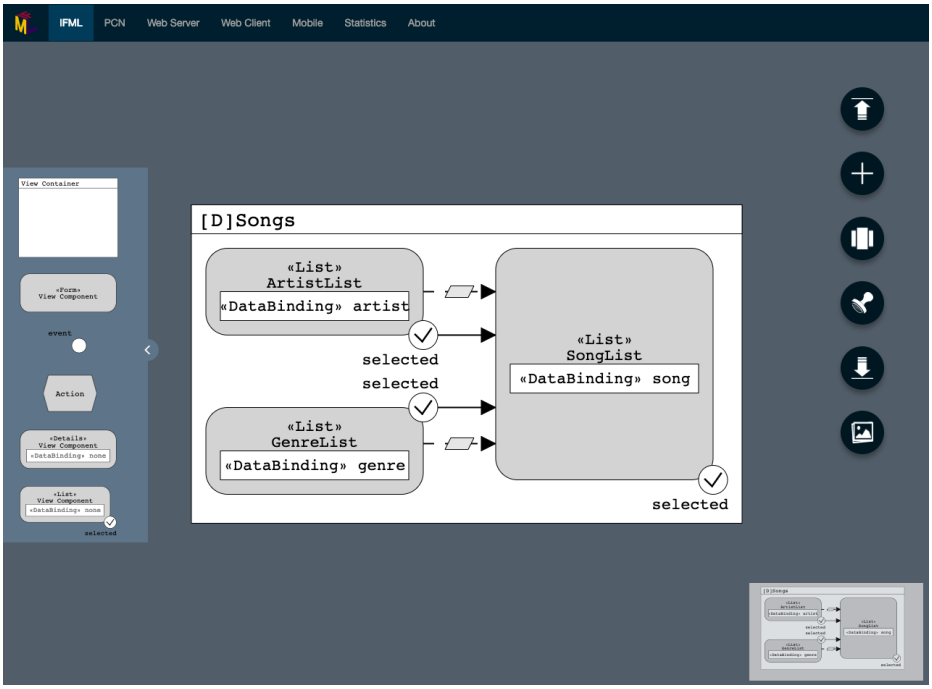


Figure 2.8: IFMLEdit.org - Model Editor

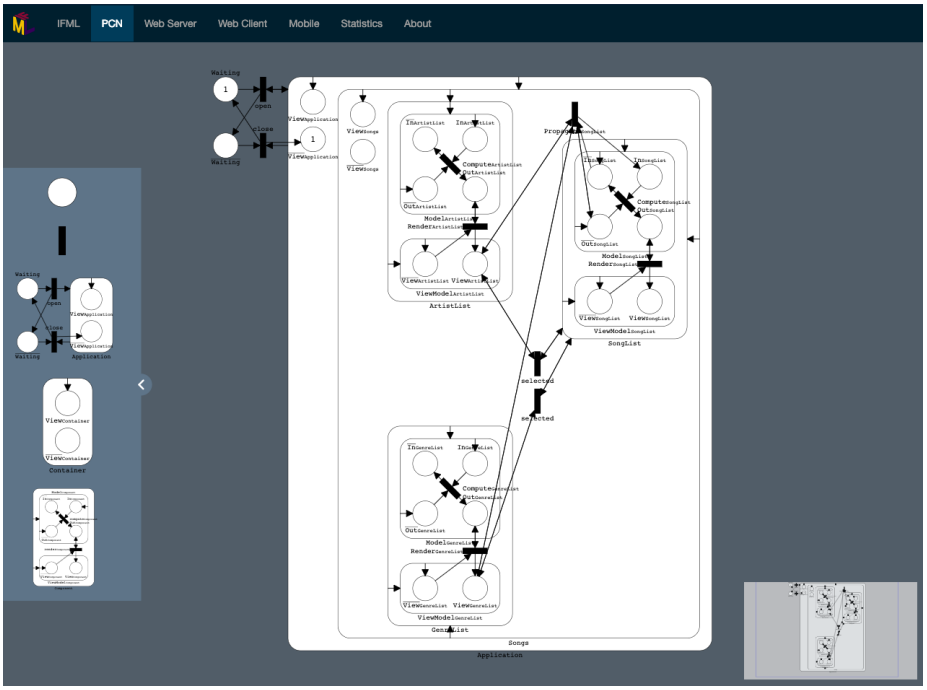


Figure 2.9: IFMLEdit.org - PCN Generator and Simulator

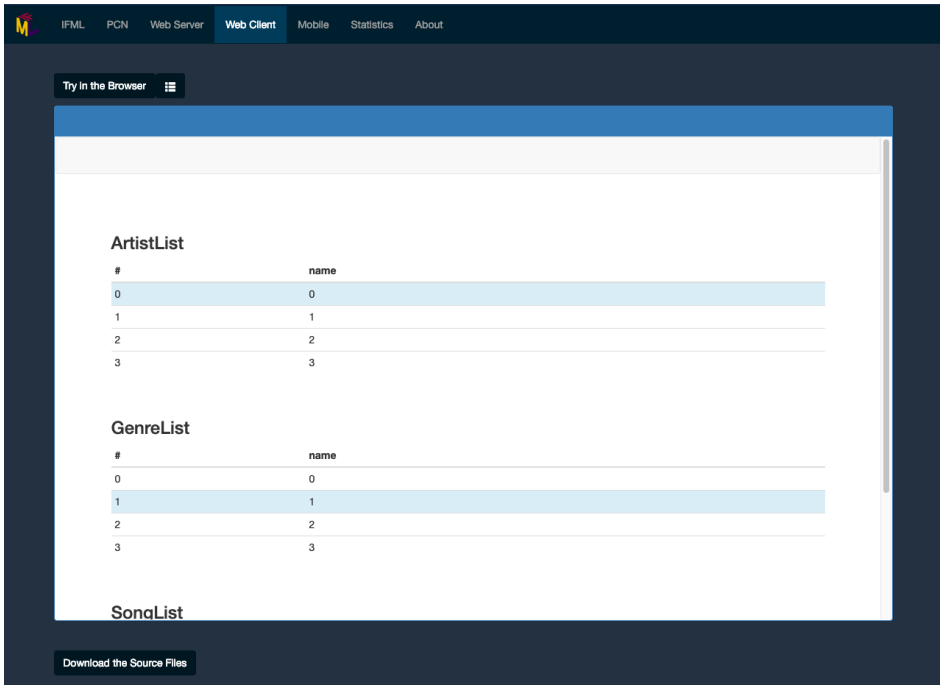


Figure 2.10: IFMLEdit.org - Web Emulator

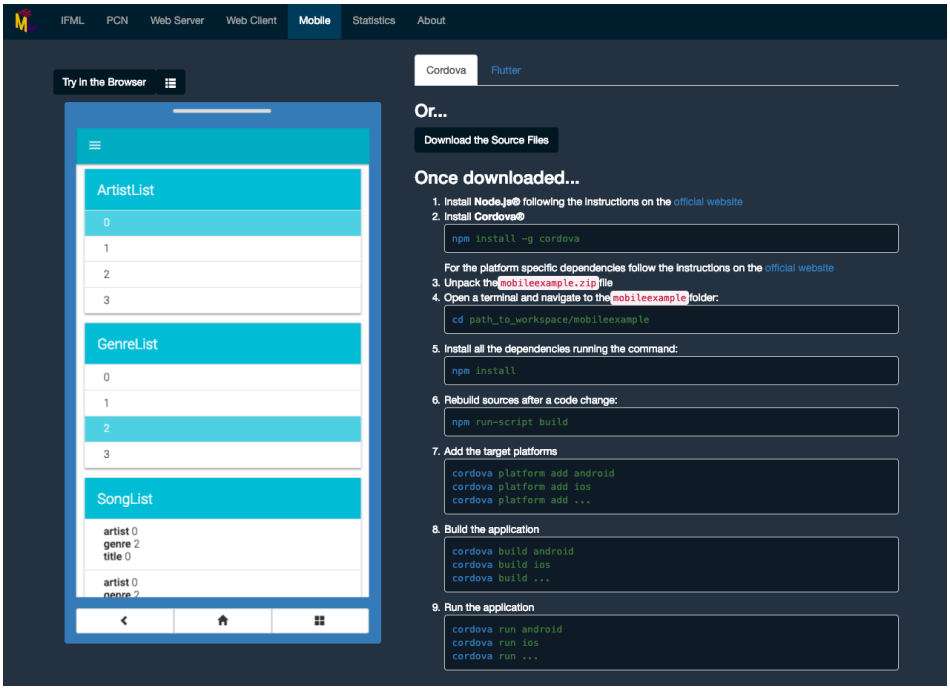


Figure 2.11: IFMLEdit.org - Mobile Emulator

2.5 Voice Recognition

In the field of Computer Science, voice recognition is defined as the ability of a machine or program to identify words and phrases in the spoken language, convert them to a machine-readable format and execute operations. Although efforts to recognize and understand the human language through a machine were conducted since the late 1800s, only the last decade has known the explosion and the widespread diffusion of this potentially revolutionary technology, mostly in terms of *voice assistants*, software able to interpret human sentences to perform tasks, interacting with real objects, online services, and human beings. The advancement of knowledge in the fields of machine learning, neural network and, more generally, artificial intelligence has allowed to obtain excellent results in terms of accuracy, reducing the error rate to below 5%, in recognition of human words and the interpretation of whole phrases.
Nowadays, global giants such as Google, Amazon, Apple, and Microsoft integrate their voice assistants in their devices and offer tools and frameworks for the development of applications that use their technology to generate services. In the next paragraphs, we will take as a model Alexa, the voice assistant developed by the research team of Amazon and used in the thesis work. In particular, we will briefly explore its system architecture and its operating principles and, in the end, we will show the main elements that characterize a skill, i.e., an application that performs operations starting from the command provided through the voice assistant.

2.5.1 Alexa System Architecture

Three components essentially constitute the system architecture of Amazon Alexa:

- An Amazon device that integrates the voice assistant (or a third-party device that can support its installation).
- The Alexa Voice Service, a cloud-based computing system that represents the brain of the architecture.
- A function or program running on the server that execute commands.

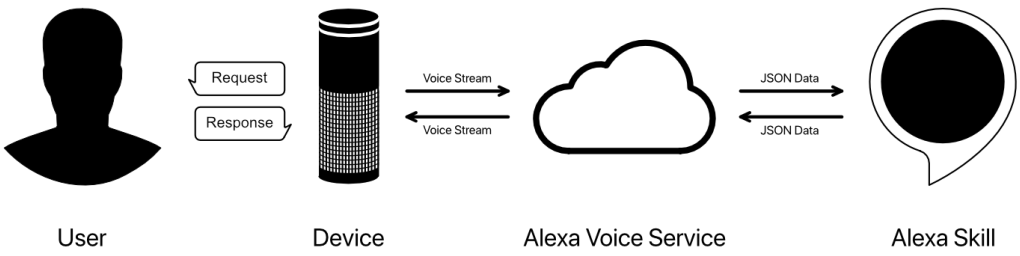


Figure 2.12: The Alexa System Architecture.

Figure 2.12 shows the Alexa system architecture, analyzed from the point of view of the life cycle of a request. To communicate with the voice assistant, the user must pronounce the so-called *wake word*, a word that orders the voice assistant to wake

up and listen to the interlocutor[3]. The words, spoken by the user after the wake word, are acquired by the software and sent to the cloud-based computing system, in the form of voice stream[4]. The Alexa Voice Service, which resides in the Amazon's servers, elaborates the received sentence, in order to interpret the intent of the user and extrapolate useful information. When the request of the user is explicit, the Alexa Voice Service inserts the critical data in a JSON object and generates another HTTP request towards the server in which runs the business logic of the application (*skill*). The information recovered from the JSON object are used to perform the operations required by the user (for example, if the user intends to book a trip within an application that lets do it through a voice assistant, the corresponding JSON object will contain the information about the date, the departure city and the destination city of the trip, needed to perform the reservation). The life cycle of the request ends returning the result of the operation to the Amazon Alexa Service, which in turn generates the response (in the form of another JSON object) that the device must emit to the user, through its speakers.

2.5.2 Alexa Skill

In order to offer great support to developers, Amazon provides a framework that allows them to create new applications, named *skills*. The paradigm behind the development of a skill revolves around three fundamental constitutive elements [43]:

- *Invocation Name*: it is the set of words that the user must pronounce to trigger the activation of the corresponding skill.
- *Intent*: it represents an action that fulfills a user spoken request. Intents can optionally have arguments called *slots*.
- *Utterance*: it constitutes a likely spoken phrase that map to an intent.

More in depth, to better understand the role each of these essential components, consider the following scenario: a user needs to organize a trip and relies on a custom skill published on the Amazon store in order to perform all the required bookings (for simplicity, we will call the skill *ThoughtlessTrip*). The skill is characterized by two *intents*: *BookFlight* and *BookHotel*. BookFlight allows booking a round-trip plane ticket. Assuming to know all the user information, the intent needs to acquire the data relatives to the departure and destination airports and the dates of the corresponding flights, i.e., parameters called *slots* that must be passed to the intent, in order to perform the task. Similarly, BookHotel defines the way through which the user can reserve a room in a hotel, for the requested period. In this case, neglecting all the possible booking variables (such as the choice of a twin or double room, *et alia*) the intent must know the check in and check out dates and the name of the hotel where the user wants to stay overnight: these parameters represent the slots of the BookHotel intent. Now, in order to execute all the reservations, the user must invoke (i.e., open) the skill. The user, in this case, must wake up the voice assistant using the wake word *"Alexa"* and pronounce the invocation name of the

[3]Indeed, the voice recognition system is programmed to acquire the words only after the dictation of the wake word. However, note that the voice assistant is in perennial listening to every word spoken by the user, as it is the only way to capture the wake word.

[4]In this regard, it is interesting to observe how, to work, the system requires that the device is connected to the internet, otherwise it is not possible to elaborate the words pronounced by the user.

application, that we hypothesize to be "open thoughtless trip". From this moment on, the voice assistant listens for a command that triggers one of the two intents implemented within the skill, and here, the role covered by the *utterances* comes into play. Utterances are phrases that manifest the *intent* (not by chance) to perform an action. During the development of the skill, the programmers define examples of sentences that a user can pronounce in order to fire an intent. These sample phrases constitute the training set of a neural network that elaborates all the possible variants, with the same meaning. As it is possible to imagine, more samples are provided at development time and more accurate will be the identification of similar phrases at run time. Example of utterances for the BookFlight intent could be:

Reserve a round-trip plane ticket from {*departure*} *to* {*destination*}*, in date* {*outward*} *and* {*return*}

I need reserve a round-trip plane ticket from {*departure*} *airport to* {*destination*}

where {*departure*} and {*destination*} constitute the slots that respectively refer to the departure and the destination airports, while {*outward*} and {*return*} define the outward and return journeys. Note how in the second sentence, unlike the previous ones, the possibility that the user provides only partial information is foreseen: in this case, since the missing data is mandatory in order to proceed with the reservation, Alexa will require to defines the dates of the two flights with a simple question[5]. At the end of the interaction, when all the necessary information has been acquired, Alexa sends the stream to the Amazon Voice service in order to proceed with the execution of the task. If the operation ends successfully, the control return to the voice assistant that listens for another intent (as for example, the BookHotel intent), and the cycle of a request restarts.

[5]At development time it is possible to establish if the fulfillment of a slot must be mandatory or can be optional. If a slot is mandatory, the framework requires to define the question that Alexa must report to the user, if the corresponding data has not been provided

Chapter 3

Related Work

The objective of the chapter is to describe the scenario in which the thesis work is inserted. More in detail, the chapter is organized into three sections that refer to the cornerstones of the proposed work, i.e., model-driven development, design patterns, and development assistance systems. The first part of the chapter (section 3.1) shows an overview of the positive and negative opinions given by the scientific community, in the field of model-driven development, presenting its potentials and limits. Secondly, in section 2.1, the focus shifts to the theories that explain the impact of patterns on software quality. In particular, we will point out the efforts aimed at automating the process of generation and coevolution of design patterns, in the context of model-driven development. Moreover, we will analyze different approaches to address the problem of pattern recognition, that works directly on models or indirectly on transformations. Finally, the chapter ends with the presentation of the methodologies implemented in order to offer assistance systems in the development of software and models (section 3.5).

3.1 MDD: the judgment of the scientific community

Ever since the branch of Software Engineering has entered the world of Computer Science, energies have been aimed at raising the level of abstraction to which developers write programs. As reported in [26], the primary goals of this infinite research lie on the attempts to significantly reduce the development efforts and the complexity of the software artifacts that programmers must handle. The first result, in this sense, goes back to the introduction of assembly language, as an abstraction over machine code. It was followed by the formalization of the third-generation languages, like FORTRAN and COBOL, and the advent of compilers that, for the first time, enabled developers to ignore registers allocation and other low-level features, letting programmers specify *what* the machine should do, rather than *how* it should do it. Object-oriented languages, such as C++ and Java, lead to additional abstractions. In each case, the achieved progress produced not negligible consequences in terms of higher quality, productivity, and expressiveness.

According to [6], Model Driven Development (MDD) can be seen as a natural continuation of this trend. Focusing on the specifications and requirements of a system, rather than the specific implementation platform, it generates models that capture the essence of an application, leaving out the development details. Atkinson and Kühne, in their work [6], emphasize how MDD can potentially improve the performances in the software development. More in detail, the paper tries to convince

the reader about the double benefits, with regard to productivity. In particular, they distinguish among:

- *short-term* productivity, measurable in terms of the number of functionality the first stable version of a software can own. The more executable functionality MDD can generate, the higher the productivity will be.
- *long-term* productivity, quantifiable with respect to how much time elapses before the released version of the software becomes obsolete. The longer an artifact stays valuable, the greater its return on investment will be, reason why it is important to reduce the primary artifacts' sensitivity to change.

In [40], Selic highlights how code automation is by far the most effective technological means for boosting productivity and reliability. In particular, he affirms that developers can attain MDD's full benefits only when it is possible to exploit its potential for automation, in terms of *automatic generation* of complete programs from models and *automatic model verification* through execution. In this sense, modeling languages take on the role of implementation languages, similar to the way in which third-generation programming languages have replaced assembly languages.
Referring to the quality of the originated models, according to [40], to be useful and effective an engineering model must possess the following five key characteristics:

- *abstraction*: a model is always a reduced rendering of the system that it represents. In the abstraction process, a good model must be able to highlight the relevant details and hide the irrelevant ones;
- *understandability*: a useful model provides a shortcut by reducing the amount of intellectual effort required for understanding. Understandability is a direct function of the expressiveness of the selected modeling language.
- *accuracy*: a model must provide a true-to-life representation of the modeled system's features of interest.
- *predictiveness*: a good model must predict the system's interesting but nonobvious properties. For instance, a mathematical model of a bridge is much better at predicting the maximum allowable load than even an exact and detailed scale model constructed out of balsa wood.
- *inexpensive*: a model must be significantly cheaper to build and analyze than the modeled system.

In conclusion, Selic asserts that a possible obstacle in the acceptance and adoption of a modeling language could be motivated by the impossibility to meet one or more of the listed criteria, during the modeling process.

Other criticisms and probable inconveniences deriving from the use of models in software development are reported in [26]. In their work, Hailpern and Tarr alert on the real possibility that final models are characterized by redundancy, in terms of components. This is reasonable if one thinks that a central tenet of MDD is to present different points of view and levels of abstraction of the same concepts. Moreover, they meditate on the fact that the more models and levels of abstraction are associated with any given software system, the more relationships are complex. This can lead to what they call a *round-trip problem*, which occurs whenever an interrelated artifact changes in ways that affect some or all of its related artifacts.

"The worst forms of round-trip problem generally appears when changes occur in artifacts at lower levels of abstraction, such as code, because inferring higher-level semantics from lower-level abstractions is much more difficult than generating lower-levels abstraction. [...] Furthermore, automatic technologies usually generate "bad" variables names because they lack a programmer's intent. Optimization technique can reorder, combine, or eliminate details that can be useful for human understanding, but are unnecessary to machine execution."

Hailpern, [26]

Finally, Hailpern and Tarr wonder if model-driven development does not move complexity elsewhere in the development process, rather than reduce it. The risk is that, increasing the abstraction level and decreasing the apparent complexity, it is possible to define relationships that are less manipulable and more difficult to understand at the code level, making debugging and maintainability prohibitive tasks.

Based on the same context of refinement and evolution of the generated system, in the aforementioned paper [40], the author reflects on how most attempts at applying automation to software modeling are limited to "power-assist" role, such as diagramming and skeletal code generation.
In other words, once the code is generated, the models are abandoned because, like all software documentation, they require expensive resources to maintain them.

In order to reduce and limit this not-negligible problem, Falzone and Bernaschina in their articles [21] and [20], present an innovative approach that promotes the reusing of models in the evolution of software systems, through ALMOsT-Git, an automation tool that tries to reapply the introduced changes in a transparent way, requiring the manual intervention of the users only when the modifications alter the application structure drastically.

Summing up what has been analyzed in this paragraph, most of the scientific community believes that, although still immature, Model Driven Development can represent a powerful and promising approach in the development of software applications. Albeit far from being a sure bet, there is more than one reason to believe that MDD has a chance to succeed in the realm of large, distributed, industrial software development.

This thesis work fully supports the belief that MDD has the potential to revolutionize how systems are developed, in terms of time, expressiveness, stability and also security. Automatize the code generation through principles such as efficiency, order, and comprehensibility can significantly reduce the development time and the efforts needed to debug and maintain the software, increasing, at the same time, the productivity. Moreover, raising the level of abstraction, MDD is able to shorten the gap existing between the stakeholders and the development team, since the model enhances understandability. One of the primary objectives of this work, therefore, is to bring a small contribution to the evolution of MDD, through the development of design patterns specifically defined to be used in the context of applications, in order to produce software from models that take into account the principle of usability and standardization.

3.2 Design Pattern Impact on Software Quality

> *"Designing object-oriented software is hard, and designing reusable object-oriented software is even harder."*
>
> Gamma, [23]

With these exact words, Erich Gamma and Richard Helm begin the first chapter of their most famous work, *Design Patterns*. Inside the book, the two authors ask themselves what are the principles and criteria that must constitute the foundations of software development and reuse, and they find a possible answer in one of the fundamental elements that constitute a cornerstone of human knowledge: the *expertise*. As reported in many articles and papers, including [16] and [1], expertise is an intangible but unquestionably valuable commodity that people acquire slowly, through hard work and perseverance.

> *"Expertise distinguishes a novice from an expert, and it is difficult for experts to convey their expertise to novices. Capturing expertise is one challenge, communicating it is another, and assimilating it is yet another."*
>
> Budinsky, [16]

In compliance with this theory, Gamma and Helm define 23 design patterns with the aim of collecting the experience gained in the field of software development and describing solutions to recurring problems, in a generic and systematic way. More in detail, they divide the design patterns in *creational patterns*, *structural patterns* and *behavioral patterns*, according to the role covered by each of them within a software system.

Over time, numerous researches have been conducted in order to investigate and understand the importance of design patterns in the field of Computer Science. Among the most critically acclaimed papers written on the subject, we mention [31] (2008) and [30] (2018), where Khomh and Guéhéneuc analyze and review (after ten years from each other) the impact of design patterns in the software quality, with respect to different attributes, such as flexibility, modularity, reusability, understandability, scalability, robustness, simplicity, *et alia*.

After having collected and investigated a large number of articles and papers related to the topic, the authors identify seven main themes debated in the literature, reporting the different point of view of the researchers, other than their personal judgment. In particular, they underline the undisputed role played by design patterns in sharing knowledge, suggesting to put in place a repository and a vetting process of patterns that have been shown experimentally to benefit software engineers and systems. According to their opinion, indeed, the construction of vocabulary would contrast the well-known *tracking problem*, represented by the uncontrolled growth of the pattern number and described in [2] by Agerbo and Cornils.

Finally, they highlight the extreme utility of design patterns both in forward and reverse engineering, and promote the formalization of the design patterns through

specification and documentation, in order to clarify and outline where and in which context a particular design pattern is more useful and more suitable than the others.

In this work, we make extensive use of patterns, sure that they represent a fundamental tool for the generation of applications that take into account the user experience principles gained over the years, in terms of page navigation, data entry, and content management. In this sense, we do not refer to the well-known design patterns developed by Gamma *et al.*, but to those presented in the book [15] by M. Brambilla and P. Fraternali. The primary reason for this choice lies in the fact that, in the thesis work, we introduce the concept of patterns in IFMLEdit.org, an online tool for the rapid prototyping of web applications that exploit the Interaction Flow Modeling Language (IFML) for describing the user's interactions, by means of information flows in reaction to user events [21]. The design patterns implemented by Gamma are addressed to Object-oriented programming, typical of application back-ends. On the contrary, IFML is a modeling language with a particular orientation on the development of the application front-ends. Consequently, the problems that characterize the implementation of a user interface commonly differ from the problems that characterize the implementation of the business logic behind it, and the solutions to that problems reside on different patterns, specially designed for the development of web and mobile applications.

3.3 Automatic Design Pattern Generator Tools

From the point of view of the implementation in the development process, efforts have been addressed in the automatic code generation of design patterns. In [16], Budinsky *et al.* propose a web-based tool that allows one to select a design pattern, customize the components, view the diagram representation and generate the corresponding code in C++. Although the Budinsky tool allows users to select design trade-offs when implementing the design model, it does not offer the ability to display the custom design model. Conversely, in their [34], David Mendoza and Michael Hall introduce S.C.U.P.E (Santa Clara University Pattern Editor), a desktop editor that permits to graphically customize a pattern in UML format, before transforming it in Java source code. More in detail, the tool provides much assistance in design patterns implementation. However, it does not provide any useful help facility to software designers, especially to novices because the help menu only describes information on the release and the copyright. Another example of an automatic code generator is given by J.Q. Huang, the developer of Designer's Assistant Tool [27]. Unlike the others, this tool uses formal methods to describe the design patterns, and it is able to generate code for different programming languages (C++, Java, and Smalltalk). Nevertheless, it does not support visualization after the design patterns have been customized, nor does it provide help facility. Finally, in [1], Admodisastro and Palaniappan promote γ-cdt, a code generation tool that tries to embrace all the strengths of the previously presented tools, limiting their defects.

In order to support the generation of applications that make extensive use of patterns and facilitate the construction and insertion of the latter within the developed models, in the thesis, we build a automatic pattern generator tool inside IFMLEdit.org, with the same principles and scopes of the code generator tools previously presented

and analyzed, but with significant differences. First of all, instead of code, this generator is built to create model instances of patterns[1]. Secondly, the insertion of the generated pattern instance inside the partial model of an application is automatic and does not require to perform copy and paste operations, as opposed to the code generated for example by the tool developed by Budinsky, that requires the manual insertion of the code inside the system. Finally, in addition to being widely customizable, the patterns components can be modified even after being created: a missing feature in the aforementioned code generation tools.

3.4 Model Driven Design Pattern Recognition

In the field of reverse engineering applied to design patterns, software tools have been proposed that could spot the use of a pattern within a software system and check if a pattern is applied correctly. In [13], Blewitt *et al.* present a pattern specification language, SPINE, that allows patterns to be defined in terms of constraints on their implementation in Java, and a proof engine called HEDGEHOG, able to process these SPINE constraints to detect pattern instances in the code. However, only 16 of the 23 design patterns developed by Gamma can be mapped into SPINE constraints, limiting the scope of applicability to two-thirds of the whole. In [35], Niere *et al.* propose FUJABA, a semi-automatic code-level tool support for pattern detection based on a recognition algorithm which works incrementally rather than trying to analyze a possibly large software system in one pass. Nevertheless, even this method presents a relatively high error rate, probably due to the low level of abstraction where it is applied. Motivated by this intuition, Hong Zhu *et al.* try to raise the level of abstraction at which pattern detection is performed and in [47] they promote LAMBDES-DP, a software tool that uses a first-order logic (FOL) language (systematically derived from the abstract syntax of UML and developed by the same Zhu and Bayley [8], [9]), in order to recognize instance of patterns directly from UML models. More in detail, LAMBDES-DP integrates two other software tools: StarUML, a UML modeling tool that allows the user to edit a diagram and export it in the XMI format that LAMBDES-DP uses as input, and SPASS, an automated theorem prover for FOL with equality, used to map the XMI file into conjectures and determine if they respect the conditions imposed by one or more of the design patterns. The limitation that characterizes this approach is mainly caused by the possible overlapping or sharing of components among different patterns, within the model (a problem that in the experimental phase led to 22% of false positives). Finally, we mention the work by Wenzel and Kelter [45] where the authors introduce a different approach, based on a different algorithm known as *SiDiff*. Starting from the *GNU diff* tool, used to compare textual documents, they elaborate the *SiDiff* algorithm to work directly on software models and to return a similarity value (from 0 to 1) between a model instance and a pattern template, without performing any transformation in other formats. However, since the difference calculation can only operate on the level of the given model type, the difference tool cannot provide absolute proof of the detected pattern instances. It rather checks which elements fulfill the defined requirements and provides hints to the developer, about where

[1] only after the whole generation of the application model, this model is transformed into code by the framework. It represents a great advantage as it is possible to define the connections and dependencies of the patterns generated with the other parts of the model, before generating the application code.

candidates of pattern instances might be found in the software model. The developer still has to verify the candidates to separate actual instances from false candidates.

In the thesis work, we implement a basic algorithm that uses the tagging principle to identify the components of the model that must match a given pattern. Although this algorithm does not present any form of intelligence and automatism in pattern recognition (it simply check that all the components of the researched pattern are present in the form of tagged elements inside the model and are properly connected among them), the use of labels makes it possible to obtain not negligible benefits, in terms of:

- precision and accuracy: the error rate is potentially reduced to 0;
- usability: the components of a pattern developed through the generator tool (implemented in IFMLEdit.org and explained in detail in the next chapter) integrate by default the corresponding tags when inserted into the model. Therefore, no user intervention is necessary in order to detect the pattern. Anyhow, the user has the total control of the components and the patterns: he can deliberately add and remove tags, or create new elements and connect them to each other, tagging any component and manually generating a pattern, that can be easily recognized by the system.
- overlapping: each tag of the model can be characterized by multiple tags. In other words, each component of the model can belong to multiple patterns at the same time, without this affecting the recognition of each of them.

3.5 Assistance Systems in Software Development

The evolution of knowledge and the dizzying technological development achieved in the last few years in the field of machine learning and neural networks have allowed the creation, insertion and diffusion of smart assistance systems within the software, in order to help users and make the user experience even more intuitive and satisfying.
In the particular context of natural language processing, much research has been conducted with the aim of offering assistants able to process and understand the human language (spoken or written). As a matter of fact, language is the preferred form of communication for expressing requirements and issuing orders. It requires little effort and allows users to enclose a meaning in a simple sentence. In conclusion, the introduction of smart systems able to interpret the human language can positively revolutionize human-machine interaction. In [18], Douglas and Russel expose a method and apparatus for editing documents through voice recognition. More in detail, the developed system is able to understand orders issued by the users and divide it into two portions, i.e., a command and a target, which respectively represent the action and the document location where it has to be performed. However, the set of orders the editor can use is limited to simple editing, zooming, highlighting and moving functions that can replace the use of the mouse.
Conversely, in the paper by Subramaniam *et al.*, the authors present the Use Case Driven Development Assistant Tool (UCDA), a software that exploits a natural language parser in order to build the use cases and class diagrams, starting from textual

specifics. In particular, the assistant tries to recognize the complex sentences and simplify them to analyze their structure (in terms of subject, verb, nouns, adjectives, *et alia*) and extract the actors, properties, methods and relationships, with the purpose of generating the models in Rational Rose (according to the Unified Modeling Language standards).
Finally, in [5], Arnold *et al.* design VocalProgramming, a system that promises to help develop code through voice commands. The aim of the project stems from the commendable desire to meet the needs of programmers with motor disabilities or those affected by injuries such as carpal tunnel syndrome.

In the thesis work, we use the expressive power of Alexa, the voice assistant created by Amazon research team, as a starting point to implement a system that guides the developers in the modeling building process in IFMLEdit.org. Therefore, the work represents one of the first attempts to integrate Model Driven Development with voice assistant tools.

Chapter 4

Designed Solution

The chapter aims to describe and discuss the main idea behind the proposed work, and the architecture of the designed solution. Initially, the chapter exposes the essential principles that characterize the thesis work (section 4.1). Then, in section 4.2, the focus shifts to the presentation of the implementation choices made to satisfy the requirements and analyzes in detail the architecture produced.

4.1 Main Idea

By carefully analyzing a wide range of applications present on the market and offering heterogeneous services, we can notice common features. For example, almost all the applications in which the user can perform operations and persistent changes at the model level, require the identification of the user himself, that has to be registered inside the platform. Similarly, the purchase of a product or the collection of a long list of information takes place through progressive steps and not through the compilation of a single huge form. Moreover, the organization of the contents inside an application is usually based on the subdivision of the items in categories and subcategories. Finally, the search for content is frequently accomplished through an advanced search tool, that normally integrates filters.
These adopted common criteria are the results of the knowledge gained in software development, from the unending research of solutions to standard problems, aimed at improving the quality and the performances of systems and applications. In other words, they represent design patterns, i.e., something comparable to the treasure of the experience.

Design patterns are accredited structures that allow building software more understandably and efficiently, under the principle according to which it is not useful to reinvent the wheel.
More and more companies, in the modern and dynamic commercial context, offer the creation of customized software, through the aggregation and connection of validated logical modules (think about the roles covered by the CRM). The benefits are tangible: the applications are created and put on the market in a short time, and the quality is guaranteed by the fact that they are the result of the assembling of tested components.

In the same way, in the context of model-driven development, design patterns can speed up the process of generation of models and applications, with the quality provided by their experimented usage.
From a technical perspective, we can distinguish among two different typologies of

design patterns:

- patterns developed to increase the quality of the applications, in terms of speed in the execution of operations and the understandability and structurability of the source code. This typology of design patterns refers to the back-end of the applications, i.e., the business logic. They are transparent with respect to application users, but they have a reflection in terms of their satisfiability because if the operations that the users try to perform inside the application are completed in a short time, the reliability of the system increases positively. From the point of view of the developers, instead, their use enhances the degree of maintainability of the software.

- patterns designed to improve the usability of a system, in terms of ease in the processes of navigation, content research, data entry, *et alia*. This typology of design patterns involves the front-end of the applications, and it has a significant impact on the overall user opinion about the software.

In this thesis, we concentrate the attention on the development of the latter type of design patterns, as we work on the IFMLEdit.org framework, an online modeling editor that uses the Interaction Flow Modeling Language for the rapid prototyping of web applications. In this context, although the editor generates the code of entire systems, the front-end covers a major role in the modeling process, since IFML is a language oriented to the definition of the interactions among the user and the components of the user interfaces.

The primary beneficiaries of front-end design patterns are unavoidably the users. As reported in [28], the degree of usability of software can be described through five significant components:

- *Learnability*, that represents how easy it is to perform basic tasks for new users.

- *Efficiency*, in terms of time required by the users to find what they are looking for.

- *Memorability*, related to how hard it is for users to repeat a performed task.

- *Error Rate*, that measures the frequency of errors committed by the users.

- *Satisfaction*, that expresses the comfort level registered by the users during the usage.

Front-end design patterns, in this sense, offer standardized ways to navigate and research contents, fill in forms and execute operations. Therefore, their usage represents a great advantage in terms of learnability, efficiency, and memorability, since from the first use of the corresponding application, users have the sensation of moving within a system that they already know and where they can guess how to orientate and operate. Consequently, the error rate committed by users decreases and the gratification increases.

Starting from the above considerations, the idea at the core of the designed solution has been to integrate an advanced support tool for the generation of design patterns at the modeling level, within the IFMLEdit.org modeling editor.
All the implementation efforts have been aimed at achieving the followings objectives:

- to significantly reduce the time and the efforts required to build a model.
- to generate applications that take into account the results obtained in the field of the user experience.
- to assist the developer in the whole building process, in terms of guidance and facilities.

In the next section, we will analyze in depth how we reached the proposed intents. In particular, we will see how it has been possible to accomplish the first two goals, through the implementation of an automatic generator of design patterns, and we will describe how the realization of the last goal has represented the more complex and delicate task since it has required the usage and the training of a smart assistant able to understand the human language, in order to perform commands.

4.2 Proposed Approach

Starting from the analysis of standard modeling frameworks, such as IFMLEdit.org, we noticed that most of them are lacking in terms of modeling support. In other words, they provide the developer with a modeling editor, assuming that the developer knows the functionality covered by each component that constitutes the metamodel of the modeling language used, other than the criteria and the principles that define the best practices relative to the insertion of new elements and the creation of meaningful relationships and connections. Although these are common assumptions that characterize all programming editors, whose purpose is to offer a tool to write code (and possibly compile it, but not teach it or suggest how to write it), in the context of Model Driven Development we are faced with a level of greater abstraction, which allows a better expressiveness and the possibility of interacting with the developer in an entirely new and smart style, closer to the human way of thinking. In a sense, it is possible to exploit this expressiveness to make the software capable of advising and expressing the best design choices to the developer, supporting it in the whole generation process.

In order to accomplish these goals, we initially introduced the concept of design patterns within the IFMLEdit.org framework, with the intent to offer basic assistance in the development of models.

4.2.1 Automatic Design Pattern Generator

The creation of an automatic design pattern generator represented, therefore, the first step in the creation process of the proposed support system. Automatizing the production of validated parts of models, indeed, it would have been possible to realize models more efficiently, both in terms of efforts and time, increasing, simultaneously, the quality of the final result.

Adapting our implementation on the characteristic style of the pre-existing modeling editor within which the pattern generator had to be integrated, we developed the tool inside a modal window [1], accessible at any moment from a menu button present

[1] a modal window, in the context of the web user interfaces, is a graphical control element subordinate to the main window of the application. It defines a way to disable the main window and concentrate the attention on the content of the modal that becomes visible in front of it.

in the main window.

The design pattern generation process has been organized into two interconnected phases. At the opening of the modal window, the first visible screen shows the list of the design patterns that the developer can generate. For each element in the list, it is possible to read a brief description of the corresponding pattern and observe a preview of the related model (Figure 4.1).

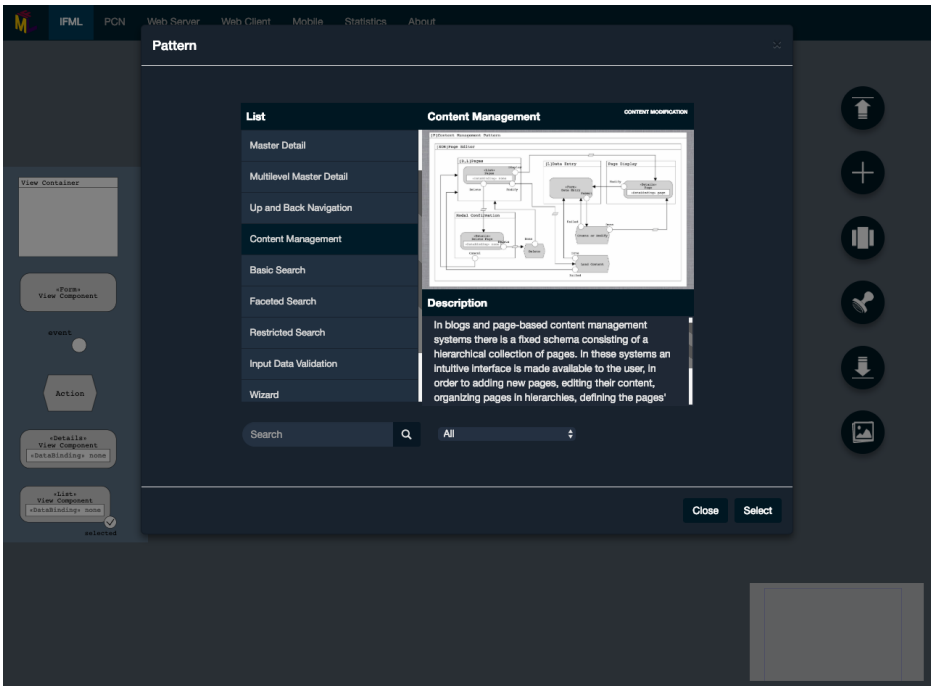


Figure 4.1: Screen of the selection phase of the automatic design pattern generator tool

After the selection phase has been completed, the modal displays the screen relative to the pattern settings, which represents the customization phase. The content of this screen change from pattern to pattern, according to the elements that compose the selected pattern and that must be configured. Figure 4.2 shows an example of customization phase, relative to the Content Management pattern.

This pattern addresses the management of the entire life cycle of a set of objects, including the ordinary operations of creation, visualization, modification, and deletion. The Content Management customization panel is divided into three sectors, respectively named Objects List, Data Entry and Object. In the Objects List sector, the developer can give a name to the collection of items and the fields that will be visible for each element that will compose the corresponding List ViewComponent. In the Data Entry sector, instead, the developer can define the fields of the Form ViewComponent that will be generated to define the properties of new elements (or to modify those of the existing ones). Finally, in accordance with the choices reported by the developer in the Object sector, a corresponding Details ViewComponent will be created in order to visualize the properties that will characterize a selected object in the list (in line with our personal design preference, the proper-

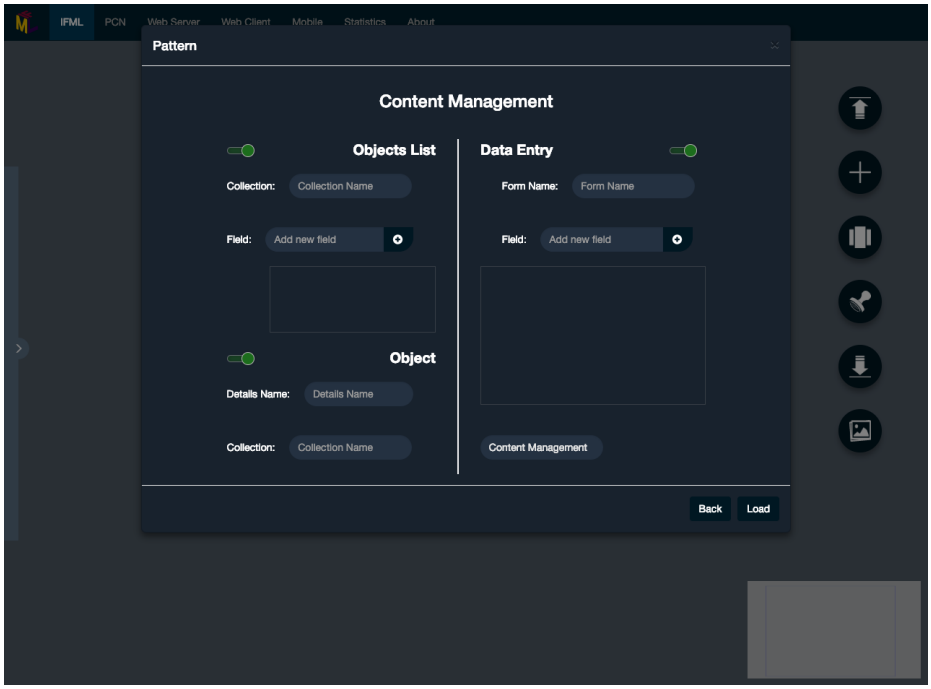


Figure 4.2: Screen of the customization phase of the Content Management pattern in the automatic design pattern generator tool

ties will be the same of those defined in the Data Entry sector). Note that, the deletion of an object is a default functionality and does not require any customization.

Figure 4.3 shows a concrete example of Content Management pattern customization and generation. The particular context of the example concerns the storage of products inside a warehouse. The information system that must manage the stocks requires to have a list of all the products in the catalog and sold by a company. For each product must be visible the details of its properties and the number of stocks inside the warehouse. Furthermore, the system must allow the insertion of new products in the database and the elimination of the products no longer present in the catalog. In order to accomplish these requirements, the developer designated for the realization of the application model can use the automatic design pattern generator tool, select the Content Management pattern (that satisfies all the exigencies) and configure it, according to the described particular context. The reported image shows an example of customization and generation of the corresponding pattern. Moreover, it highlights the relations among the sectors of the configuration panel of the generator tool and the corresponding components of the generated model of the pattern.

Depending on the selected pattern, it is possible to choose from different degrees of flexibility in the model to be generated. For example, concerning the same Content Management pattern, the needs of the developers could be different, in different applications where even the role of final users changes. Assuming that the developer wants to create a wish list within a user profile (where the user can save and manage the products he intends to buy), it is clear that once the user has inserted a product in the list, he can only display the details of the product, remove it or select another product from the list. The user cannot create new products or modify the properties

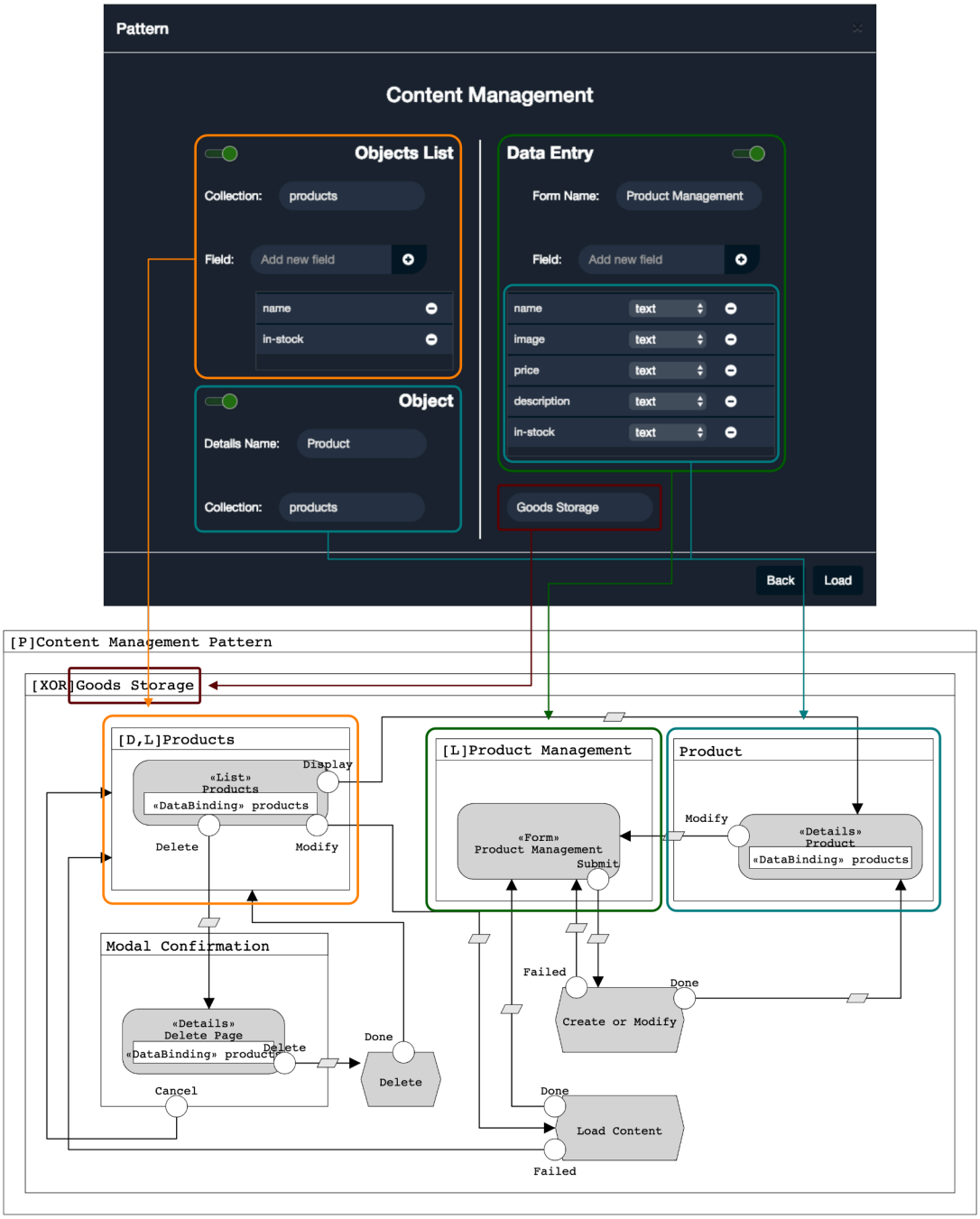


Figure 4.3: Mappings from the customization components to the generated pattern components of the Content Management pattern

of the products present in the wish list because he has not the permissions to do these operations. For this reason, the configuration of the Content Management pattern allows the flexibility to remove components of the patterns that are not useful in specific contexts. Referring to the proposed example, the Data Entry sector is not useful and can be disabled through a corresponding switch button (visible, for instance, in Figure 4.2). However, the flexibility of any pattern is limited to constraints, in order to maintain the meaning structure of the pattern [2].

In the end, as reported in [39], design patterns can be characterized by structural and dynamic parts. The structural part defines the essence of the pattern, while the dynamic one represents those pattern components that can change in different contexts, without compromise the meaning of the pattern itself. This is the case of the Wizard pattern, that supports the partition of an extended data entry procedure into logical steps that must be followed in a predetermined sequence, increasing the order and the readability and reducing the compilation efforts. In this circumstance, the aim and the structure of the pattern are explicit, but the number of steps in which the data entry must be divided is not known a priori. For this reason, the automatic design pattern generator implements functions that allow to programmatically generate the dynamic part of a pattern, starting from the template of its structural part (that is static and does not change).

The list of the design patterns that it is currently possible to create through the implemented automatic design pattern generator is reported in Appendix A[3].

4.2.2 Pattern Detection and Updating

In IFMLEdit.org, the traditional workflow imposes the insertion of new elements and the following configuration of their properties through a modal edit window. Consequently, the customization of the model occurs progressively, element by element, and it is a long process that requires many efforts. Moreover, it is a mandatory operation to give a sense to the model and make it easily readable and understandable. Each element requires a unique identifier that distinguishes it from the other elements in the model and a name which consents to the reader an immediate perception of its role inside the model. However, when a new element is generated, the editor assigns to it a pseudo-random identifier and a default name that is equal to any other element of the same type (such as, "ViewContainer", "Action", "Event", *et alia*), requiring the developer intervention every time.

The automatic design pattern generator tool implemented in the framework partially resolves this problem in the configuration phase, where the developer is required to name the components of the selected pattern in a unique panel, while the identifiers are automatically deduced from the given names[4]. Anyhow, when the pattern instance is inserted into the model, any future change returns to being at the element level,

[2] In the Content Management pattern, for example, it is not possible to disable more than one sector. Otherwise, the entire pattern lost its meaning.

[3] All the presented pattern copy and take inspiration from the work by M. Brambilla and P.Fraternali in [15]

[4] the algorithm that defines the identifiers performs the appropriate checks to verify that no duplicates are produced

thus losing the benefits introduced by the generator.

For this reason, we worked on the implementation of an algorithm that allowed the developer to recognize patterns within the model, to continue making changes at the pattern level, recycling the same configuration panel used at the time of its generation.

The basic idea behind pattern research and identification is that any design pattern owns a unique imprint, given by the components that characterize it, other than the relationships and connections that they establish with each other. However, the development of a model is a non-deterministic process since, a priori, we do not know how many components the model will contain and how the components will be connected. So, we are facing a significant problem whose resolution algorithm could requires a lot of time at run time or, potentially, never end. Moreover, some design patterns represent the evolutions of other patterns and, in a sense, they own the same imprint of the ancestors, plus subtle differences that may not be caught, if the algorithm is not powerful enough. In this case, the risk is that the algorithm recognizes only parts of the pattern that lead back to the basic versions of it, rather than the evolutions.

Shifting the problem from a global perspective of the model to a local one does not represent the solution since any sub-model (represented, for example, by the content of a ViewContainer) constitutes by itself a model whose dimensions are unknown. In this context, we can only rely on the ViewComponents and the Actions that are the only elements that cannot be the parents of other elements (excluding the Events) in the hierarchical relations, representatives of the IFML models.

During the feasibility study of the algorithm, we identified two possible approaches to solve the problem. The first approach suggested the creation of immutable areas, represented by a ViewContainer inside which only and exclusively all the components that would have constituted the pattern could have belonged to it. This solution would have guaranteed a targeted and fast search but would have imposed limits on the construction of the model, significantly reducing the expressive power of the modeling language. In fact, no other element could have been inserted into a ViewContainer in which a design pattern would have been defined. Conversely, the second approach involved the use of the tagging principle, according to which each element that would have belonged to the pattern would have owned a label with the name of the pattern of which it would have been part. In this context, when the automatic generator would have produced an instance of design pattern, it would have marked each element with the tag, without requiring any intervention by the developer. Moreover, any element could have been inserted into the ViewContainer containing the pattern, since during the search and identification process, having no tag or (having a different one), it would not have been considered as part of it.
For these reasons, in the end, we decided to opt for the latter approach.

In the implemented algorithm, the elements that compose a design pattern are generated inside a ViewContainer which represents a sort of box of the entire pattern. This ViewContainer is labeled as the *root* of the pattern while the contained ViewComponents represents the *nodes*. A *node* can lie only inside a *root* ViewContainer, marked with the same pattern tag. Whenever the developer wants to make changes at

the pattern level, he can select the *root* ViewContainer and push an icon button that starts the research of the pattern components inside it. If the pattern is found, the editor notifies the developer that the search produced a positive result and opens the modal window that represents the configuration panel of the design pattern (in order to perform all the necessary changes). Otherwise, the editor informs the developer that no matching was found.

Figure 4.4 sums up, in a single image, all the essential concepts behind the algorithm and the process of pattern detection. The pattern in question is the same Content Management pattern exposed in details in subsection 4.2.1, to which other components have been added later (the elements highlighted in red). The orange ViewContainer, marked with the [P] symbol, represents the *root* of the pattern. The green components inside the *root*, instead, represent the *nodes*. After selecting the *root* element, the developer push the icon button highlighted in yellow, in order to launch the search algorithm. The latter recognizes the type of design pattern to look for, from the corresponding tag of the *root* element. Therefore, respect to the supposed example, it directs the research towards the primary components of the Content Management pattern, that are:

- a List ViewComponent that represents the set of objects to be managed in the application.
- a Details ViewComponent that displays the objects that the user wants to remove during a delete operation.
- a Form ViewComponent that allows the creation of new objects.
- a Details ViewComponent that visualizes the properties of a selected object.

More in detail, the algorithm tries to navigate the components, using the NavigationFlows which connect ViewComponents to other ViewComponents or Actions, through the Events. In the specific case of Content Management Pattern, it verifies that, starting from a List ViewComponent, exists a path that connect it to a Details ViewComponent, then to an Action, and finally returns to the ViewContainer that contains the starting List ViewComponent (the orange flow in the image). Similarly, it checks the blue, red and purple paths in the figure. The presence of other connected elements such as the ViewComponents and the Action marked in red, does not influence the research since they are ignored, not being tagged. At the end of the algorithm, the editor notifies the result of the search to the developer, that in this given case is a positive matching (Figure 4.5), and opens the configuration panel (like the one presented in Figure 4.3, but in the reverse process).

4.2.3 Voice Assistant

The integration of the automatic design pattern generator and the algorithm of pattern detection and updating, in the IFMLEdit.org framework, represents a significant improvement in support of the modeling process, both in terms of time and quality of generated models. However, this tool does not offer, in itself, active assistance in real time: the tool is available to the developer, who can decide in which circumstances

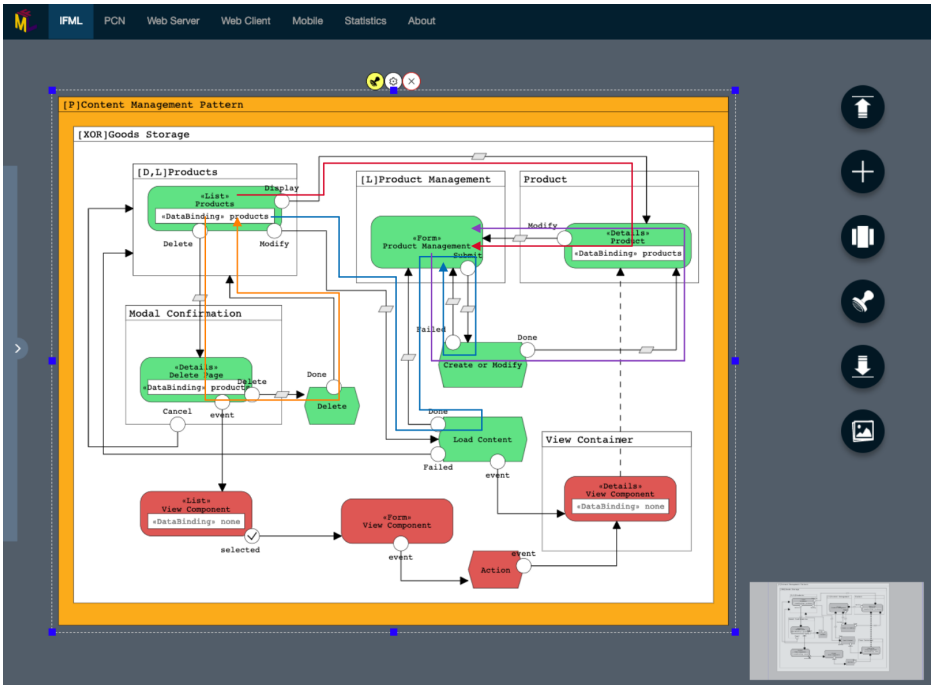


Figure 4.4: Content Management Pattern Detection Algorithm

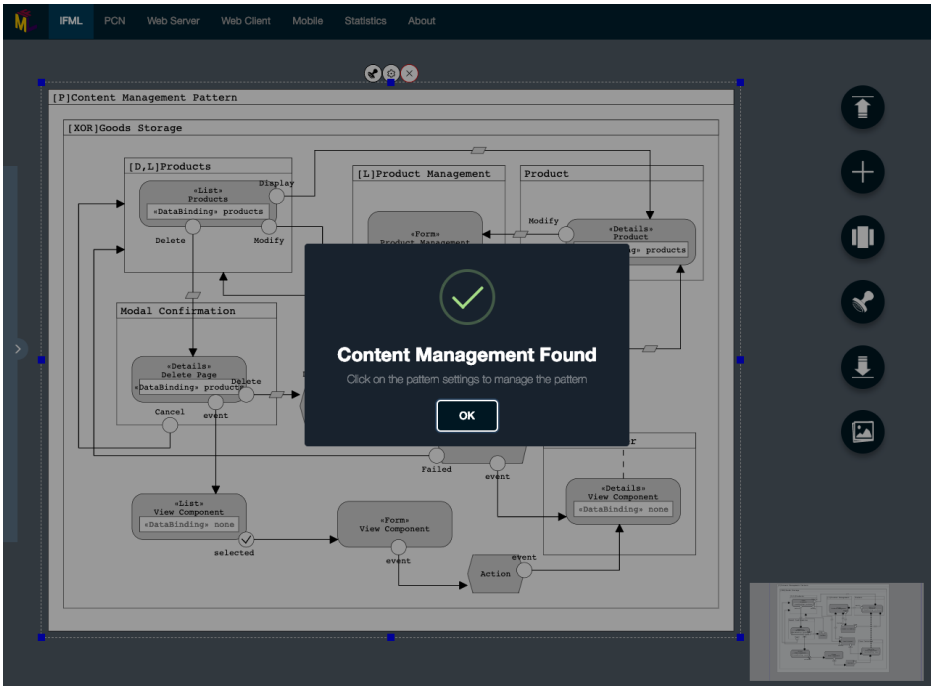


Figure 4.5: Content Management pattern search with positive matching

use it to generate patterns, but the developer still represents the only mind that can think of how to design the model and what decisions to make during the building process. As a result, this implemented solution can only assume a passive role inside the model editor.

Therefore, the next objective of the thesis work have been directed towards the introduction of a form of artificial intelligence that could transform a passive support into an active assistance, able to communicate and guide the developer, understanding his commands, providing advice and building models automatically, starting from specific needs.

The research and the progress in technology have led, in recent years, to the creation of neural networks and machines capable of understanding verbal language and interacting with human beings, through their preferred means of communication, i.e., the voice. The rapid spread of voice assistants is a foretaste of how the market for smart applications and devices will evolve, in the near future. They represent the simplest and most humane way to issue orders and commands to smart objects connected to the network, reducing the efforts of users requesting services and simultaneously allowing them to perform other actions, in parallel.

In the particular context of Model-Driven Development, a voice assistant can support the developers with a different level of expertise, in a non-invasive manner. More in detail, it can assume a priority role, taking the responsibility to generate model components programmatically (according to the requirements expressed by the developer), or merely supervise the developer in the building process, with tips and revisions.

In the thesis work, we integrated Amazon Alexa Voice Assistant inside IFMLEdit.org, through the implementation of an Alexa Skill, that we named *Model Creator*, able to understand commands expressed by developers and communicate with the model editor to perform operations. The skill has been developed in order to address two main goals:

1. to offer a complete assistance system in the automatic realization of complete IFML models, starting from a brief interview to the developer, with the aim to understand the application specifications.

2. to provide advanced support in the realization of models, in which the developer is the real leading actor of the building process and the assistant represents the servant that executes the commands imparted.

In the next paragraphs, we will analyze in depth the way in which the developer converses with the voice assistance and the Model Creator Skill exchanges information with IFMLEdit.org to fulfill the presented purposes.

4.2.4 Model Creator Skill

Model Creator is an Alexa Skill developed in the Alexa Developer Console (Figure 4.6), a suite made available by Amazon to define the *intents* of applications that make use of the voice assistant to execute tasks.

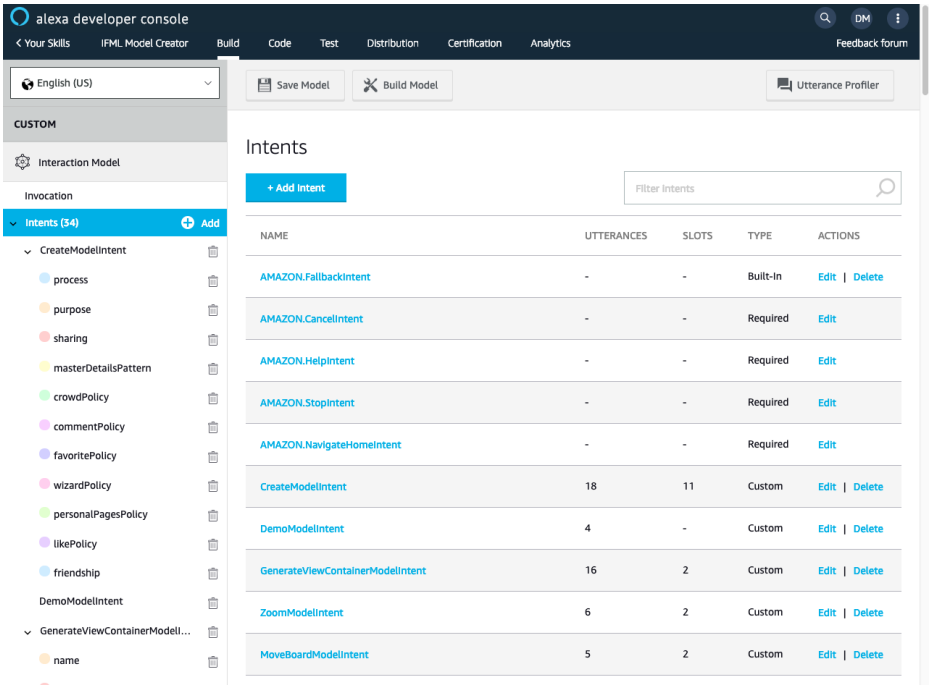


Figure 4.6: The Alexa Developer Console

The business logic of Model Creator lies on IFMLEdit.org, and the communication process between the skill and the framework is represented by the following information flows:

1. The developer wakes up Alexa and pronounces the invocation name to open Model Creator.

2. Alexa collects the voice stream and sends it to the Alexa Voice Service that recognizes the command, opens the application and notifies IFMLEdit through a JSON Object, sent inside an HTTP request with destination the IP address of the server in which the online framework runs.

3. The received JSON Object is parsed, and the logic consequent to the skill opening is executed.

4. The framework communicates the result to the Alexa Cloud Service, that, in turn, elaborates the voice answer stream and sends it to Alexa.

5. Alexa uses the speakers to communicate the opening state to the developer and awaits the formulation of new commands.

6. For each new intent, the process is repeated: the voice stream is sent to the Alexa Voice Service that identifies the intent and forwards the acquired information to the framework. The latter executes the required operations on the model editor, and the reverse response process is performed.

The whole communication process is summarized in Figure 4.7.

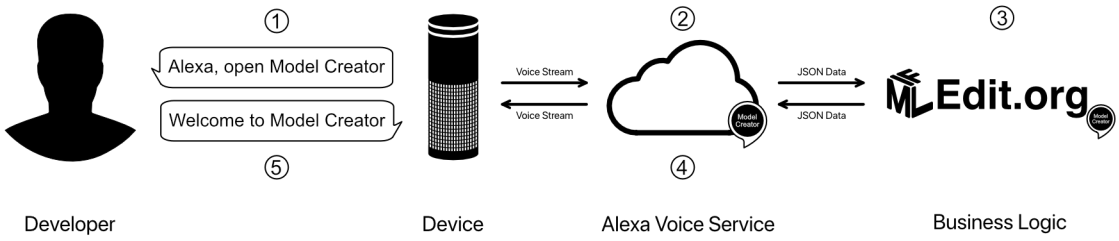


Figure 4.7: Communication process in Model Creator

An intent can be represented by a single interaction or a dialog composed by multiple requests and answers. In the first case, the developer asks for the execution of a simple command, while, in the second case, the request is complex and the voice assistant demands the progressive acquisition of information. The reason is that any intent can require the fulfillment of mandatory and optional parameters. When the number of parameters is high, it is difficult to provide all the required values, by means of a single sentence. Moreover, some parameters may become mandatory, depending on the value assumed by other parameters. Therefore, the interaction is divided into multiple steps[5].

In order to offer different levels of assistance to the developer, Model Creator implements two different typologies of support. When the developer shows the intention to generate a model through the supervision of the voice assistant, the skill requires to know if the developer intends to be guided in the entire modeling process or proceed with advanced support.

4.2.4.1 Guided Models

The development of guided models is explicitly intended for all those cases where:

- the customization of the application is not a priority task.
- the developer wants to develop standard application models quickly.
- the developer is not sufficiently experienced to build models independently.

In this context, the voice assistant assumes the role of coordinator and holds the reins of the design process. Through a multi-turn dialog, it progressively acquires all the information required to define a complete model (that makes uses of the patterns defined in the thesis work) and, at the end of the interaction, it demands the framework to load the model which best fits the developer needs (inside a collection of predefined templates).

In the thesis, we provided the possibility to build four different typologies of guided models, that map to the following categories of applications:

- E-commerce platforms, aimed at the sale of commercial products online.

[5]During the dialog and depending on the state of the information acquisition process, the framework can take control of the dialog with the developer, soliciting the request of particular information (i.e., parameters values), or delegate the formulation of the next questions to the Alexa Voice Service.

- Social Network applications, for the publication of posts and the creation of virtual relationships.
- Blogs, for the publication of articles and the sharing of experiences and knowledge.
- Crowdsourcing systems, with the purpose of collaborating with other users on the realization of tasks.

During the interaction process, the voice assistant, initially, requires to know the purpose of the final application that the developer wants to build (Figure 4.8).

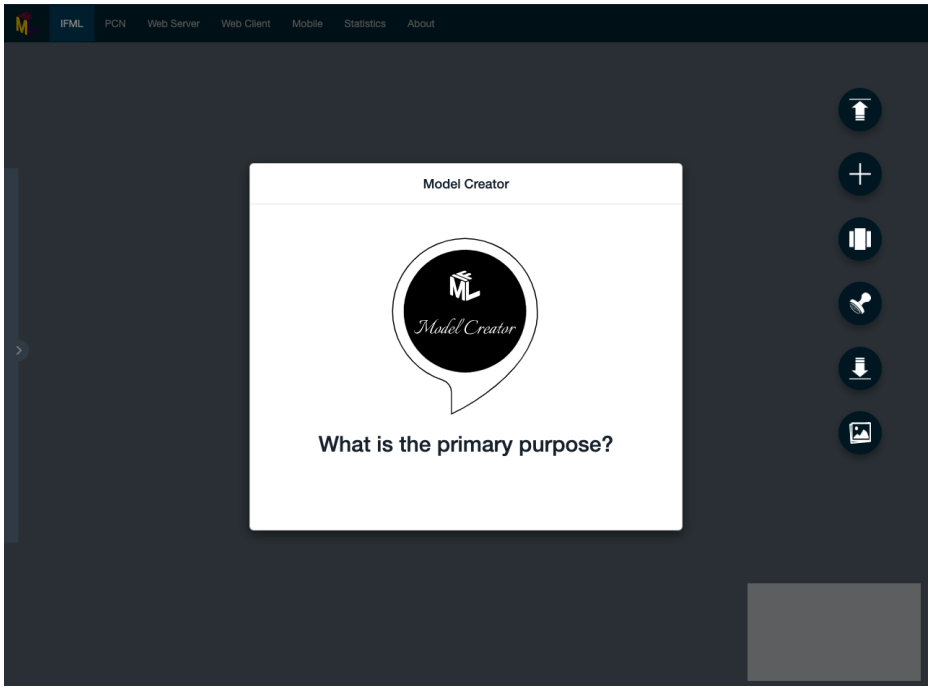


Figure 4.8: The voice assistant requires to know the purpose of the final application

Consequently, the voice assistant changes the formulation of the next questions, addressing the dialogue towards the ultimate intent of the user. Figure 4.9 shows the state of the interaction during the acquisition of the information needed to design the model of an e-commerce application: the modal window displays the current step of the process, the question posed to the user, and the data already collected. At the end of the interaction, the assistant demands the editor to load the model that best fits the user's specifications (Figure 4.10).

The complete technical guide, that describes all the steps of the interaction and all the possible flows is reported in the Appendix B.2.

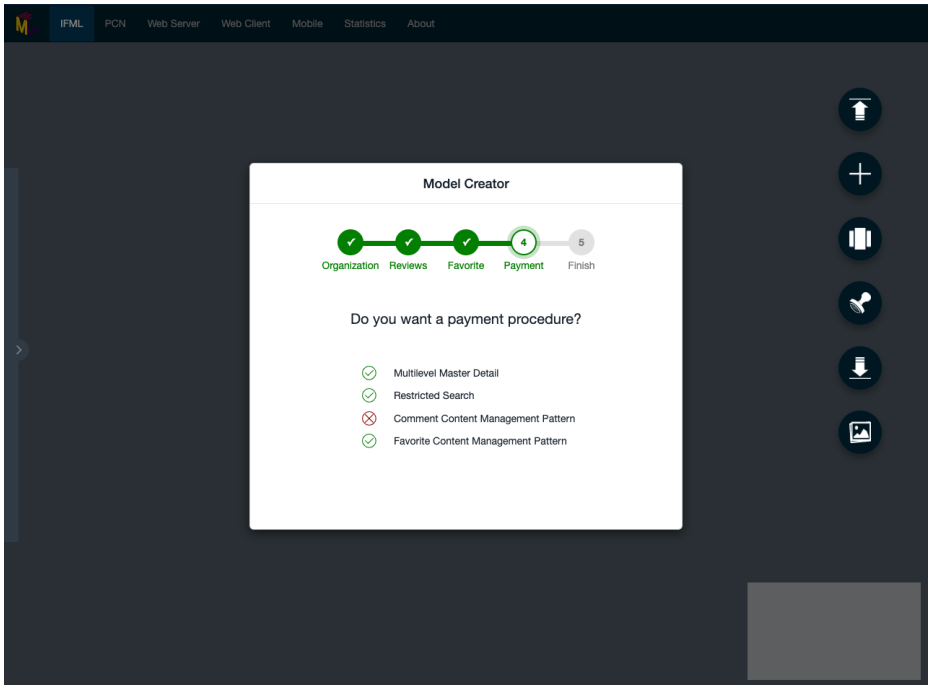


Figure 4.9: The modal window displays the status of the information acquisition process in IFMLEdit.org

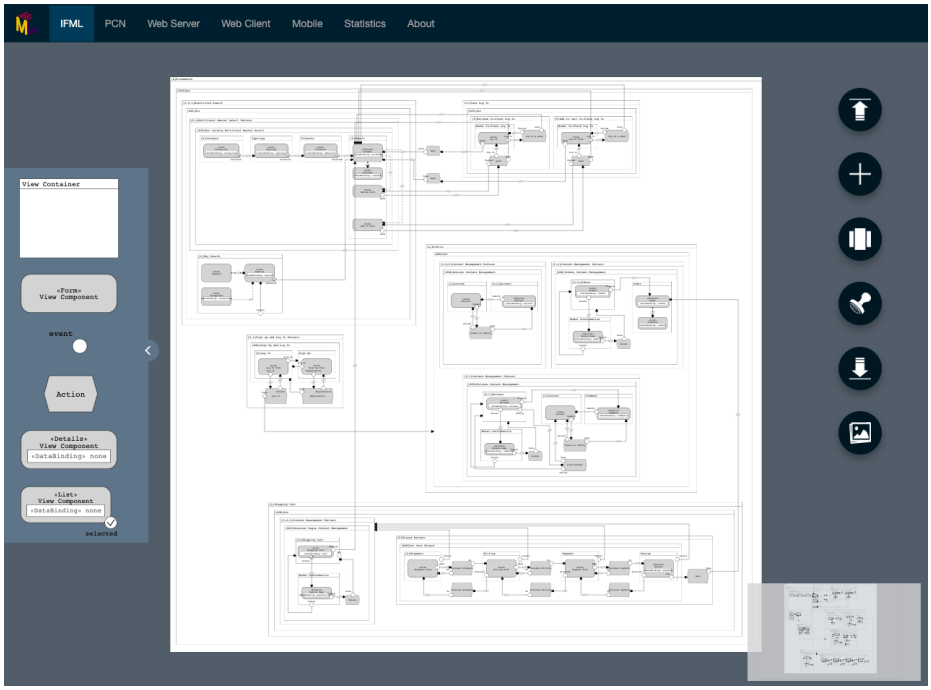


Figure 4.10: At the end of the interaction the model is generated in the editor

4.2.4.2 Advanced Support

The advanced mode is designed especially for all those situations in which:

- the application requirements are well defined and unconventional.
- the developer is an experienced model maker.

The voice assistant listens to the new orders given by the developer, who can control the entire modeling process through voice commands. In this way, for example, the developer can generate a model at the same time as he is reading a specification document, accelerating the production and reducing the efforts. The developer can require the insertion of new elements, manage their personalization, move components, create nesting, add design patterns and define connections between elements, without using the keyboard and the mouse to control the model editor.

Figure 4.11 shows a situation in which the Goal ViewContainer and the Goal Form ViewComponent inside it must be inserted into the Internal ViewContainer, to the right of the Internal Form ViewComponent.

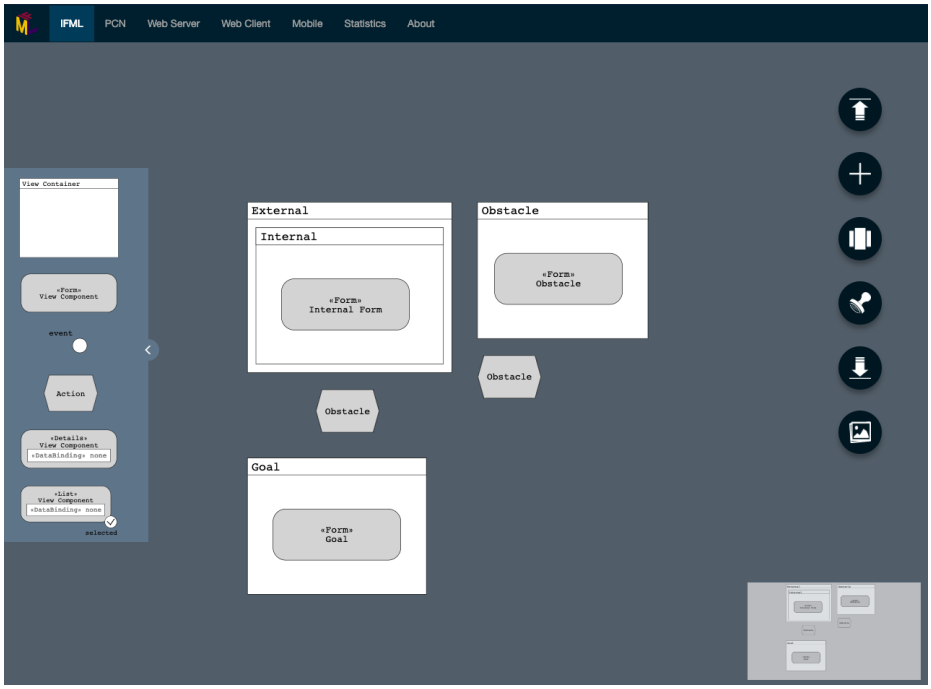


Figure 4.11: The state of a model, before the insertion operation

In this case, the Internal ViewContainer fits perfectly inside the External ViewContainer. Moreover, the latter is surrounded by other elements which represent obstacles to the enlargement of the External ViewContainer. The command that orders to carry out the requested operation is the following:

Insert ***goal viewcontainer*** *inside* ***internal*** *view container,*
to the ***right*** *respect to* ***internal form***

where the words highlighted in bold represent the slots (i.e. the values of the parameters) of the utterance that fire the intent. In order to perform the required operation, the algorithm implements the following steps:

1. computes the new positions of the Goal ViewContainer and the Goal form ViewComponent contained inside it.
2. calculates the new position and the new dimensions of the External and Internal ViewContainers, so that the Goal ViewContainer can be contained inside them.
3. computes the new position of the obstacles elements.

The result of the insertion operation is reported in Figure 4.12

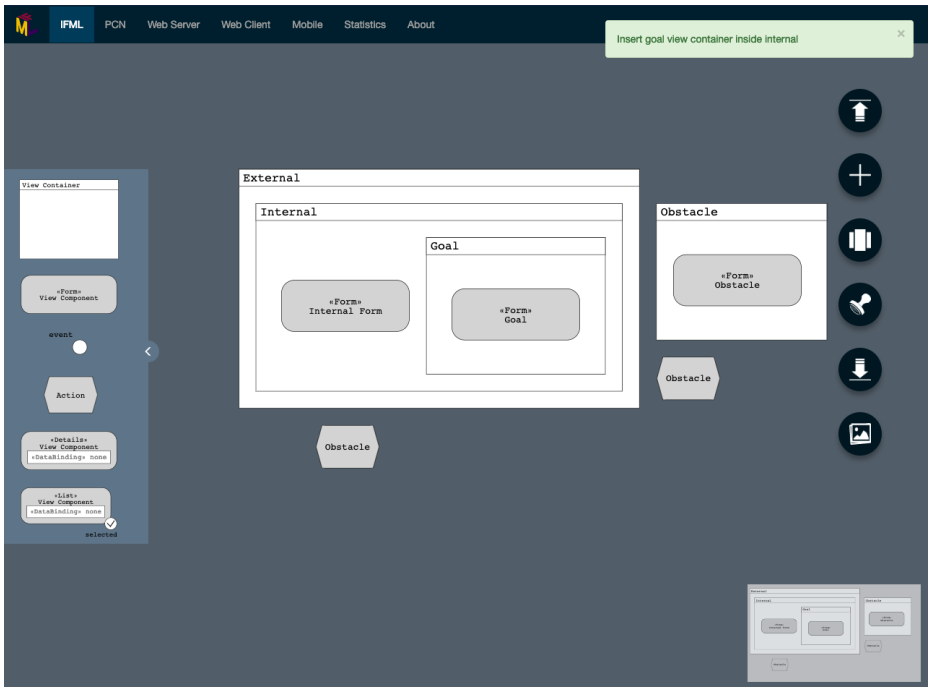


Figure 4.12: The state of the same model, after the insertion operation

The complete list and description of all the advanced commands implemented in the skill to control the development process using the voice assistant, is reported in the Appendix B.3.

Chapter 5

Experiments and Validation

Experiments were conducted throughout the thesis, in order to validate the implemented concepts. In this chapter, we report the most significant case studies that have allowed us to verify the quality of the work performed, highlighting its strengths and showing its limits.
In section 5.1, we will compare the realization of models without the use of the automatic design pattern generator tool (described in the section 4.2.1), against the realization of the same models with the support of the automatic design pattern generator tool. We will offer an estimate in terms of the number of operations required in both cases, underling how this statistical approach can be mapped in the time domain.
Subsequently, in section 5.2, we will analyze in depth one of the most complete models that can be built through the guided voice assistant, allowing us to discuss about different qualitative metrics (such as flexibility, duplication of components and accessibility), respect to which it is possible to evaluate the work done. The chapter ends with outline considerations about how the developed infrastructure can significantly reduce the gap between the stakeholders and the developers of applications, and how the integration of a vocal assistant can represent a valid support in terms of accessibility, allowing people with injuries or motor disabilities to be able to design models in a simple and intuitive way (section 5.3).

5.1 Performance Analysis

The benefits, deriving from the implementation of the automatic design patterns generator and the integration of the voice assistant as support of modeling operations, are tangible but difficult to be statistically quantified. Finding metrics that objectively prove that the work done is useful, it is not a simple task since the quality of the models is mostly based on:

- the satisfaction of the developers during the modeling process, with regards to the efforts needed to design new models.
- the judgment of the developers who find themselves having to work with the code generated from the models, both in terms of understandability and maintainability.
- the level of gratification of the final users of the resulting applications, in terms of usability, simplicity of research and learning, rate of errors committed while browsing content, *et alia*.

Moreover, the use of design patterns tends to bring a gain that can be subjectively higher or lower depending on the degree of experience of the modelers, the developers, and the users.

A metric that, more than the others, can give an objective estimate of the work done is time. Indeed, fixed some parameters, it is possible to analyze whether the use of design patterns and the support of a voice assistant has a significant impact on:

- the modeling times.
- the development and maintenance times of the generated code.
- the completion times of tasks during the usage of the application.

The realization time of a model depends on the degree of knowledge of the tool and the modeling language. We should, therefore, analyze how the times vary according to the approach used, depending on the type of developer (newbie or expert) involved in the generation, and observe whether the temporal gain is constant, or cannot be considered independent with respect to the degree of experience. In our case, it was not possible to refer to a sufficiently large and varied sample of developers in order to obtain consistent statistics. In the same way, it is necessary to verify that the final application is used by users with different levels of mastery in the world of Web and mobile applications, analyzing how the usability gain varies according to the complexity of the tasks to be performed. Finally, we must consider the fact that the time gain obtainable with the use of the implemented tools may depend on the degree of complexity of the constructed model and of the application that follows. Also in this case, the lack of a sufficiently broad and diversified user pool prevented the obtaining of reliable and objective results.

For these reasons, we proceeded with another type of evaluation that uses a metric that provides a rough approximation, but that can reasonably be mapped in the time domain, producing a quantitative estimate of the improvements made by the work done. The realized experiments have been completely oriented to the evaluation of the first stage of the model-driven development process, i.e., the design of the model, that represents the focus of the thesis work. In detail, we counted the number of operations needed to develop models using the automatic design pattern generator, comparing the results with those obtained building the same models without the support tool. The models used for the tests are the same that it is possible to generate through the guided voice assistance, namely, the complete models of an e-commerce application, a social network, a blog, and a crowdsourcing platform (they are shown in Figures B.11, B.4, B.16, and B.21, at the end of the appendix B). The operations we considered in the count are summarized in four categories:

- *Insertions*: refer to any addition of new elements inside the model.
- *Connections*: allude to any creation of information flows among two components of the model.
- *Customizations*: relate to any change made to a component of the model, both in terms of appearance and properties.
- *Bindings*: concern any link between fields, filters, parameters, and results belonging to two different components.

In this context, it is important to highlight how, in the absence of an automatic tool capable of recording any operation in real time, it has not been possible to count the number of errors committed and corrected during the model development, as well as all the operations of resizing, nesting and moving of the elements. The computation has been performed at the end of the process, on the ultimate models. It follows that the obtained results represent rounded down approximations of the actual number of operations required for the realization of the models. However, it is reasonable to think that the lack of these data is exclusively at the expense of the development approach guided by the automatic design pattern generation tool. In fact, by automating the construction of whole parts of the model, the number of enlargement and nesting operations needed to insert the elements that make up a design pattern, into the ViewContainers that collect them, is significantly reduced. Furthermore, the automatic generation of connections, fields, collections, and smart ids should also reduce the number of possible errors that can be committed.

Table 5.1 shows the numbers relative to the operations required to build the model of an e-commerce application, without the support of the automatic generation tool. More in detail, the table focuses the attention on each design pattern that constitutes the model and shows the contribution that each of them brings to the final result. The last row describes all the components and connections that are not fragments of any pattern but represent necessary elements of completion and aggregation among the parts that constitute the model.

Ecommerce (without)	Insertions	Connections	Customizations	Bindings	Total
Multilevel Master Detail	10	3	52	3	68
Restricted Search	7	4	34	3	48
Reviews In-Place Log In	12	3	51	9	75
Favorite In-Place Log In	12	3	47	9	71
Add to Cart In-Place Log In	12	3	47	9	71
Sign Up and Log In	16	7	71	18	112
Account Content Management	11	4	60	20	95
Orders Content Management	14	5	62	3	84
Favorite Content Management	11	5	43	3	62
Reviews Content Management	24	12	111	23	170
Personal Pages Content Management	12	5	48	2	67
Wizard	38	16	200	70	324
Support and Aggregation Elements	21	27	101	47	196
	200	97	927	219	1443

Table 5.1: Detail of the number of operations needed for each pattern in order to build the model of E-commerce, without the support of the automatic design pattern generator tool

Overall, to be generated, the model requires 200 insertions, 97 links, 927 customization changes and the definition of 47 bindings, for a total of 1443 operations.

Ecommerce (with)	Insertions	Connections	Customizations	Bindings	Total
Multilevel Master Detail	1	0	21	0	22
Restricted Search	1	1	8	1	11
Reviews In-Place Log In	1	0	5	0	6
Favorite In-Place Log In	1	0	5	0	6
Add to Cart In-Place Log In	1	0	5	0	6
Sign Up and Log In	1	0	10	0	11
Account Content Management	1	0	14	0	15
Orders Content Management	1	0	10	0	11
Favorite Content Management	1	1	14	1	17
Reviews Content Management	1	0	18	0	19
Personal Pages Content Management	1	1	15	1	18
Wizard	1	0	23	0	24
Support and Aggregation Elements	21	27	101	47	196
	33	30	249	50	362

Table 5.2: Detail of the number of operations needed for each pattern in order to build the model of E-commerce, with the support of the automatic design pattern generator tool

Table 5.2, instead, presents the same information but relatively to the development of the model with the support of the automatic design pattern generator. The insertion of each pattern, in this case, requires only a single operation. Moreover, defining the fields and the filters that composes a pattern, the generator automatically creates the corresponding parameters, results, connections, and bindings, significantly reducing the number of operations expected to customize the components and design the information flows among them. The number of operations needed to generate the support and aggregation elements, however, remains the same of the other approach, not being a responsibility of the generator.
In conclusion, the total number of operations decreases to 362, given by the sum of 33 insertions, 30 links, 249 customization changes and the definition of 50 bindings.

Tables 5.3 and 5.4 shows the results obtained computing the number of operations required to generate the model of a social network, respectively without and with the support of the automatic design pattern generator tool.

Social Network (without)	Insertions	Connections	Customizations	Bindings	Total
Master Detail	6	1	32	1	40
Basic Search	16	8	66	6	96
Sign Up and Log In	16	7	71	18	112
Account Content Management	11	4	60	20	95
Friends Content Management	16	6	57	3	82
Posts Content Management	24	12	114	28	178
Support and Aggregation Elements	35	26	146	37	244
	124	64	546	113	847

Table 5.3: Detail of the number of operations needed for each pattern in order to build the model of Social Network, without the support of the automatic design pattern generator tool

Social Network (with)	Insertions	Connections	Customizations	Bindings	Total
Master Detail	1	0	14	0	15
Basic Search	1	0	13	0	14
Sign Up and Log In	1	0	10	0	11
Account Content Management	1	0	14	0	15
Friends Content Management	1	0	13	0	14
Posts Content Management	1	0	17	0	18
Support and Aggregation Elements	35	26	146	37	244
	41	26	227	37	331

Table 5.4: Detail of the number of operations needed for each pattern in order to build the model of Social Network, with the support of the automatic design pattern generator tool

Also in this case the use of the tool shows a considerable impact in the final number of operations required: in the first case, in fact, 124 insertions, 64 connections, 546 customization changes and the definition of 113 bindings are required, for a total of 847 operations, while in the second case, the numbers fall to 41, 26, 227 and 37, for a total of 331 operations.

Tables 5.5 and 5.6 presents similar results for the blog model, while tables 5.7 and 5.8 compare the counts obtained by developing the crowdsourcing model with the two different approaches.

Blog (without)	Insertions	Connections	Customizations	Bindings	Total
Multilevel Master Detail	10	3	50	3	66
Restricted Search	7	4	34	3	48
Comments In-Place Log In	12	3	51	9	75
Favorite In-Place Log In	12	3	47	9	71
Sign Up and Log In	16	7	71	18	112
Account Content Management	11	4	60	20	95
Favorite Content Management	11	5	43	3	62
Personal Pages Content Management	24	12	108	22	166
Comments Content Management	24	12	102	19	157
Support and Aggregation Elements	15	17	66	21	119
	142	70	632	127	971

Table 5.5: Detail of the number of operations needed for each pattern in order to build the model of Blog, without the support of the automatic design pattern generator tool

Blog (with)	Insertions	Connections	Customizations	Bindings	Total
Multilevel Master Detail	1	0	19	0	20
Restricted Search	1	1	8	1	11
Comments In-Place Log In	1	0	5	0	6
Favorite In-Place Log In	1	0	5	0	6
Sign Up and Log In	1	0	10	0	11
Account Content Management	1	0	14	0	15
Favorite Content Management	1	1	13	1	16
Personal Pages Content Management	1	0	16	0	17
Comments Content Management	1	0	14	0	15
Support and Aggregation Elements	15	17	66	21	119
	24	19	170	23	236

Table 5.6: Detail of the number of operations needed for each pattern in order to build the model of Blog, with the support of the automatic design pattern generator tool

Crowdsourcing (without)	Insertions	Connections	Customizations	Bindings	Total
Multilevel Master Detail	10	3	51	3	67
Restricted Search	7	4	35	3	49
Sign Up and Log In	22	11	90	20	143
Master Account Content Management	11	4	60	20	95
Worker Account Content Management	11	4	60	20	95
Tasks Content Management	24	12	107	21	164
Tasks Input Data Validation	7	2	27	9	45
Support and Aggregation Elements	15	9	48	5	77
	107	49	478	101	735

Table 5.7: Detail of the number of operations needed for each pattern in order to build the model of Crowdsourcing, without the support of the automatic design pattern generator tool

Crowdsourcing (with)	Insertions	Connections	Customizations	Bindings	Total
Multilevel Master Detail	1	0	20	0	21
Restricted Search	1	1	9	1	12
Sign Up and Log In	1	0	12	0	13
Master Account Content Management	1	0	14	0	15
Worker Account Content Management	1	0	14	0	15
Tasks Content Management	1	0	16	0	17
Tasks Input Data Validation	1	0	6	0	7
Support and Aggregation Elements	15	9	48	5	77
	22	10	139	6	177

Table 5.8: Detail of the number of operations needed for each pattern in order to build the model of Crowdsourcing, with the support of the automatic design pattern generator tool

The results are once again promising: the number of operations is reduced from 971 to 236 in the case of the blog model and from 735 to 177 in the case of the crowdsourcing model.

Analyzing the obtained reductions in terms of percentage, we discover that, on average, the support tool can decrease the number of operations for about 71% (Table 5.9). Although it is impossible to convert this percentage into the time domain, we can in any case state with any probability that the obtainable time gain is still significant.

	Without Support Tool	With Support Tool	Percentage Reduction
E-commerce	1443	362	74,91%
Blog	971	236	75,69%
Social Network	847	331	60,92%
Crowdsourcing	735	177	75,91%
		Mean	71.86%

Table 5.9: Detail of the number of operations needed for each pattern in order to build the model of Crowdsourcing, without the support of the automatic design pattern generator tool

The number of operations required to develop models using the advanced voice assistance is of the same order of magnitude since the assistant himself uses the automatic generator to create design patterns. The guided assistance, instead, allows the developer to generate an entire model after a brief interaction of about a minute. However, this approach cannot be compared to the previous ones since, currently, it permits to generate only a restricted range of types of models. Therefore its power is limited to the available templates.

5.2 Case Study

In this section, we analyze in depth the developed model of an e-commerce application, with the aim of showing and validating, through the help of a complete example and from a qualitative point of view, the strengths and the limitations of thesis work, in the modeling process. In particular, we will emphasize how the implemented tools consent to maintain a high degree of flexibility, obtaining a structured organization of the components of a model, reducing at the same time the risk of redundancy.

5.2.1 Model Description

The model selected for the case study was created a first time, with the aid of the automatic design pattern generator, and a second time, with the support of the voice assistant. The application to which the model refers is that of an online store for the sale of commercial products. Within the application, the user must be able to search and view the products (divided into categories and sub-categories), adding them to the cart or a wish list. Each product must be reviewable. Furthermore, the user must be able to modify the profile data, analyze the status of the shopping cart and the wish list, track the orders made, and view or modify reviews written on the products purchased. Finally, the user must be able to complete an order, providing billing, shipping and payment information.

A representation of the model is reported in Figure 5.1. The model is subdivided in five areas connected to each other. The first area (Figure 5.2) is represented by a Restricted Search and Multilevel Master Detail design patterns. It maps to the section of the real application where the user can navigate to explore the products of the catalog, supported by a search tool to speed up the task.

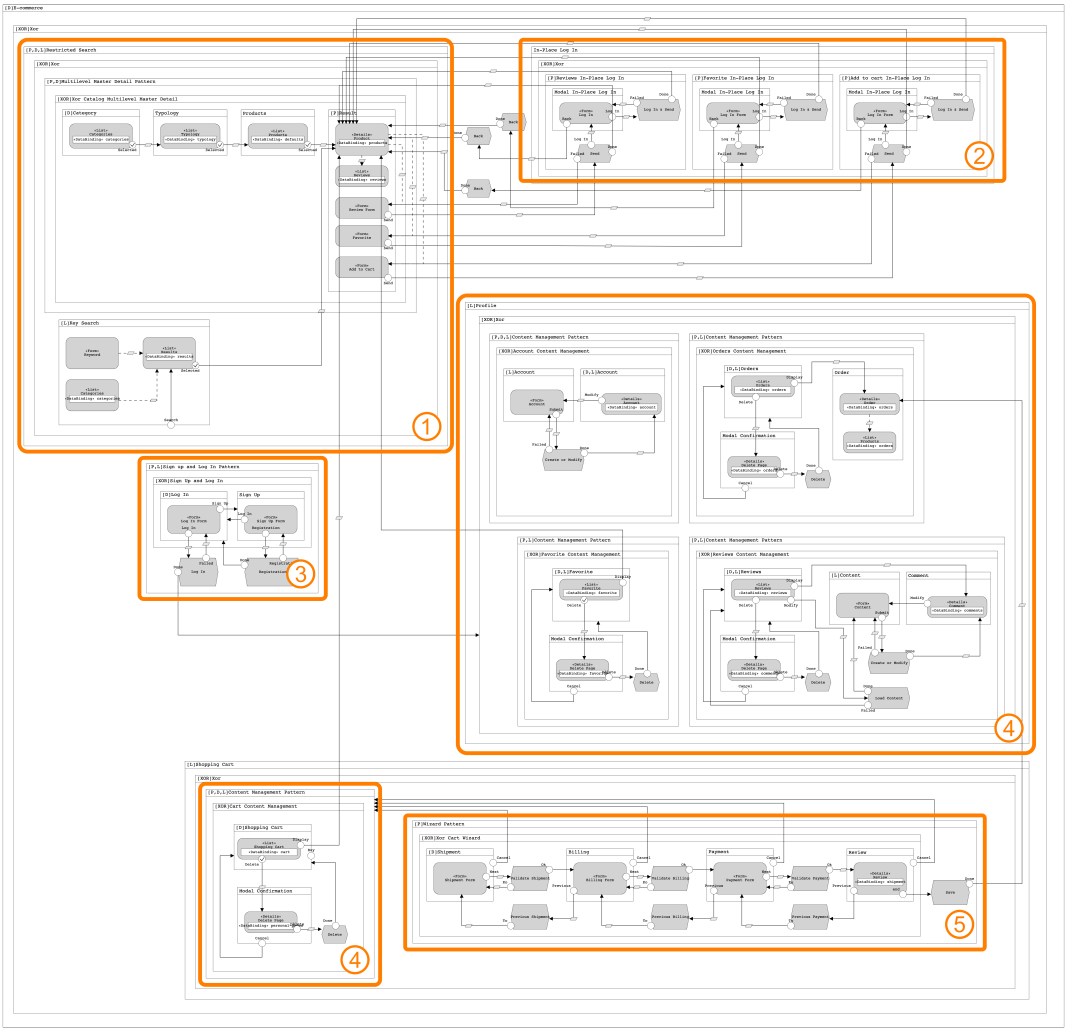


Figure 5.1: E-commerce Application Model

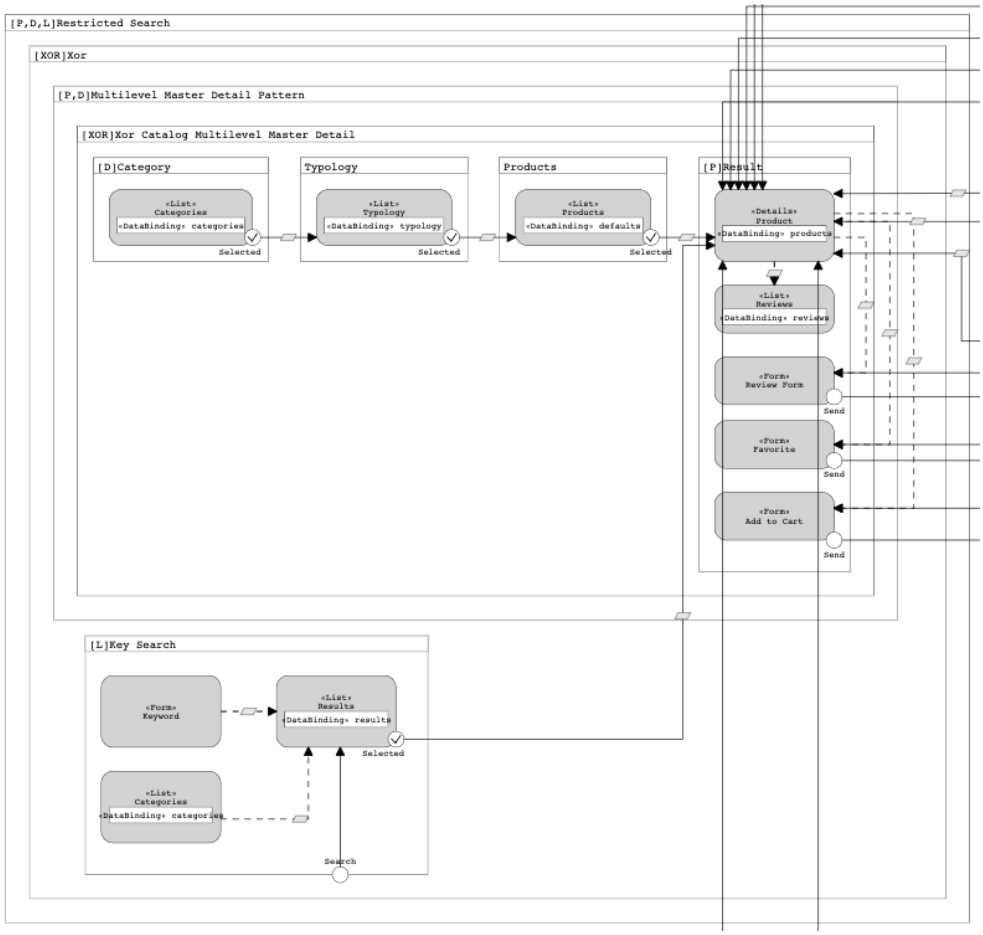


Figure 5.2: E-commerce Model - Area 1, Restricted Search and Multilevel Master Detail design patterns

The second area (Figure 5.3), instead, is constituted by the In-Place Log In design patterns that are connected to the corresponding forms of Area 1. Their role is to allow the user who has not yet authenticated, to log in "on-site", without leaving the currently active page, in which he has already entered data to send to the server (which otherwise would be lost). In the context of the model in question, three design patterns of this typology have been defined, in order to manage authentication during the operations of: review of a product (Reviews In-Place Log In), insertion of a product inside the cart (Add to cart In-Place Log In), and addition of a product within the wish list (Favorite In-Place Log In).

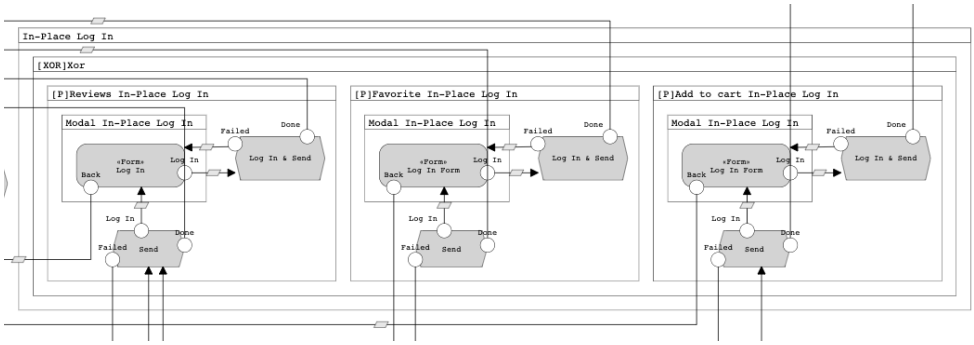


Figure 5.3: E-commerce Model - Area 2, In-Place Log In design patterns

The third area constitutes the page where the user can register or access the application (the Sign Up and Log in design pattern is reported in Figure 5.4) while Area 4 (Figures 5.5 and 5.6) defines the pages where the user can manage its profile, in terms of personal information (Account Content Management pattern), wish list (Favorite Content Management pattern), cart (Cart Content Management pattern), orders (Orders Content Management pattern) and reviews (Reviews Content Management pattern).

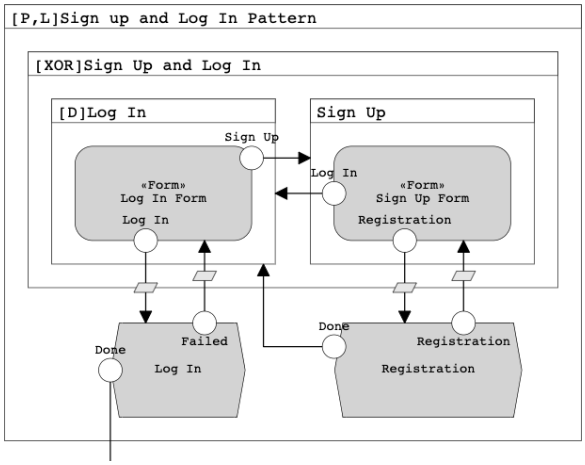


Figure 5.4: E-commerce Model - Area 3, Sign Up and Log In design patterns

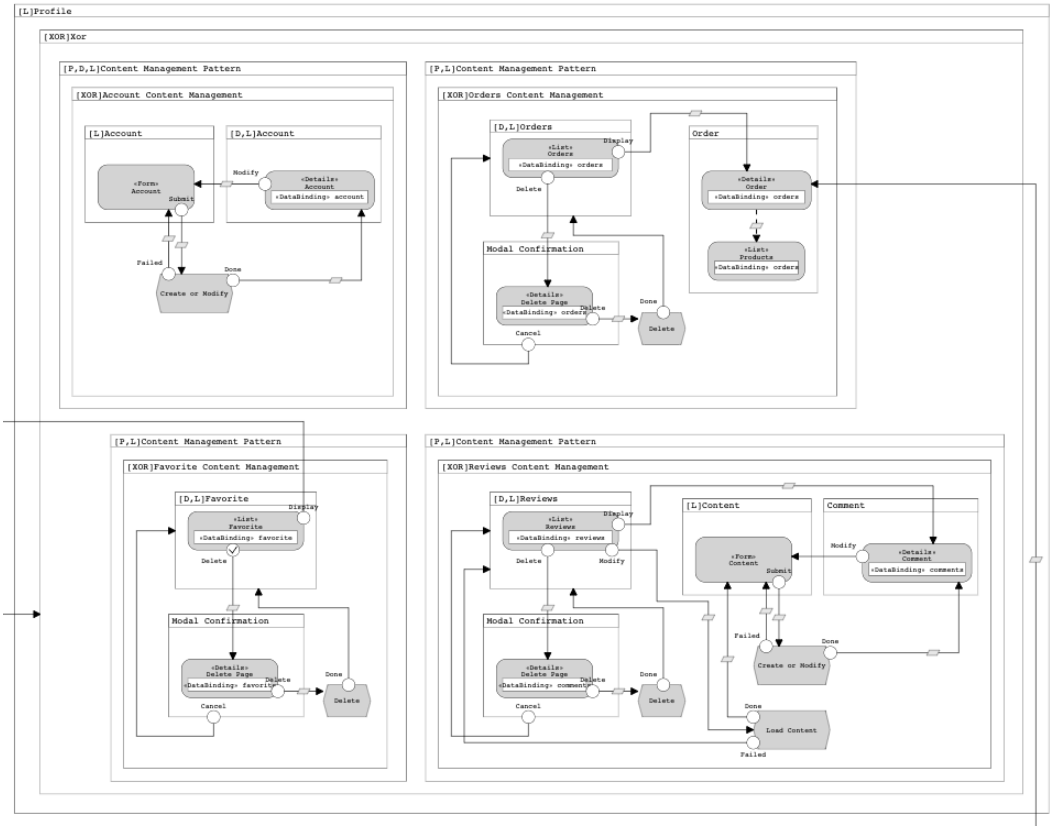


Figure 5.5: E-commerce Model - Area 4 (first part), Content Management design patterns

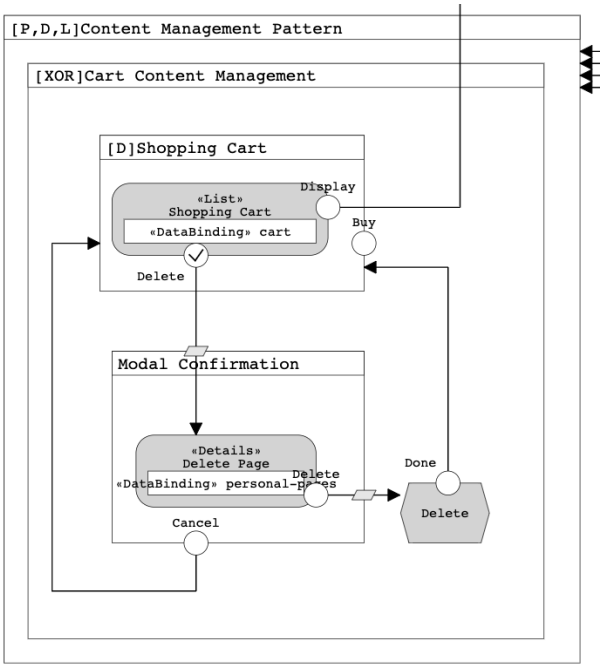


Figure 5.6: E-commerce Model - Area 4 (second part), Content Management design pattern

Finally, Area 5 defines the payment procedure that the user must follow in order to complete a purchase. It is constituted by a Wizard design pattern (Figure 5.7) that subdivide the information to be acquired into three steps (and a final recapitulation detail ViewComponent).

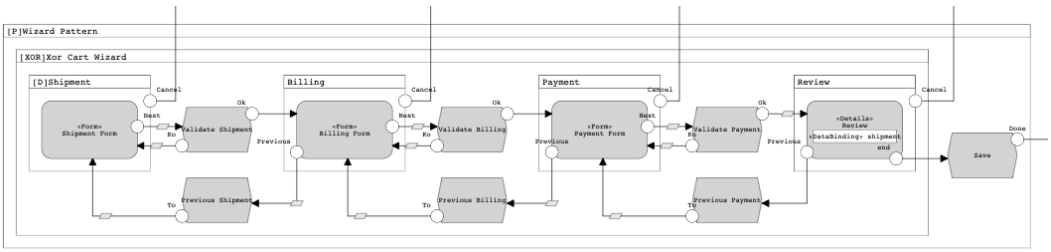


Figure 5.7: E-commerce Model - Area 5, Payment Procedure Wizard design pattern

5.2.2 Model Flexibility

The presented model is just an example of how design patterns can be used and connected to each other, within an application, in order to standardize the design process for the benefit of all the stages of the implementation and the subsequent use of the ultimate system. Although this may appear to be a limitation to the creativity and flexibility of the model itself, this problem does not arise in our context as, depending on needs and requirements, the modules can be replaced with alternative and more suitable design patterns (for example, the Multilevel Master Detail could be changed with the simpler Master detail, if the products to be shown do not need

to be subdivided into categories and sub-categories). Furthermore, since the concept of immutable area is not present within the design patterns (according to the way they were developed in the thesis work), the latter can be integrated with other components that do not belong to them (such as the Restricted Search in Area 1 of our case study, that contains the Multilevel Master Detail, which in turn contains forms ViewComponents that does not belong to it).
Through the guided assistance it is possible to develop the same model in 16 variants, depending on the answers provided to the voice assistant, during the acquisition process of the specification of the application to be built. However, given that currently the models are generated from templates preloaded within the framework, this solution presents a scalability problem. In fact, for each variable to which the user assigns a value during the dialogue with the assistant, different combinations of components correspond. Therefore, the number of models that must be preloaded is exponential with respect to the number of variables (and the possible values they can assume) of which the application model is composed. The solution to this problem could be represented by the development of a form of artificial intelligence able to dynamically and automatically generate models, programmatically inserting the patterns and connecting them together in a smart way in real time, during the multi-turn dialog among the modeler and the assistant.

5.2.3 Duplication Reduction

One of the risks concerning model-driven development is represented by the duplication of components. The standard of the Interaction Flow Modeling Language, for example, prohibits that two Events can be connected to the same Action, even if they need to trigger the same functions to complete the same task. This forces to define two Actions with the same properties.
The duplication of components can have inevitable consequences in the generated code, which can present identical blocks and structures, making maintenance more expensive and requiring more considerable efforts to prevent this from happening.

The introduction of standardized structures such as those represented by design patterns can accentuate this problem, making the risk more concrete. Also for this reason, during the implementation of the thesis work, we decided not to make the pattern blocks immutable, allowing the developer to make changes to its components, without compromising the structure. The tagging principle that characterizes the generated patterns allows checking if the components that constitute them are still present and correctly connected, regardless of which new elements have been added inside the pattern container, and which connections have been defined with their native components.

In our case study, we can analyze how it is possible to manipulate design patterns, avoiding the risk of duplicating the components and without destroying their essence. Focusing the attention on the models which represent the Multilevel Master Detail and the Restricted Search design patterns (described in details in sections A.2.3 and A.3.3 of the Appendix A) we can observe how, in Area 1 (Figure 5.2) of the proposed e-commerce model, the first is nested into the second, sharing the same detail ViewComponent. Figure 5.8 highlights in orange the components that constitute the Restricted Search pattern, and in yellow the ones that compose the Multilevel

Master Detail pattern. Finally, the detail ViewContainer marked in red represents the element in common. In order to reach this condition, the following operations have been necessary:

1. after generating the two design patterns separately, the Multilevel Master Detail pattern have been inserted into the Restricted search pattern.
2. the detail ViewComponent of the Restricted Search pattern has been deleted.
3. the selection event of the Restricted Search pattern has been connected to the detail ViewComponent of the Multilevel Master Detail.
4. the shared detail ViewComponent has been tagged as belonging to both the Multilevel Master Detail pattern and the Restricted Search pattern.

In this way the structure of the two patterns have been preserved and the algorithm of pattern detection is able to recognize them even after the modifications.

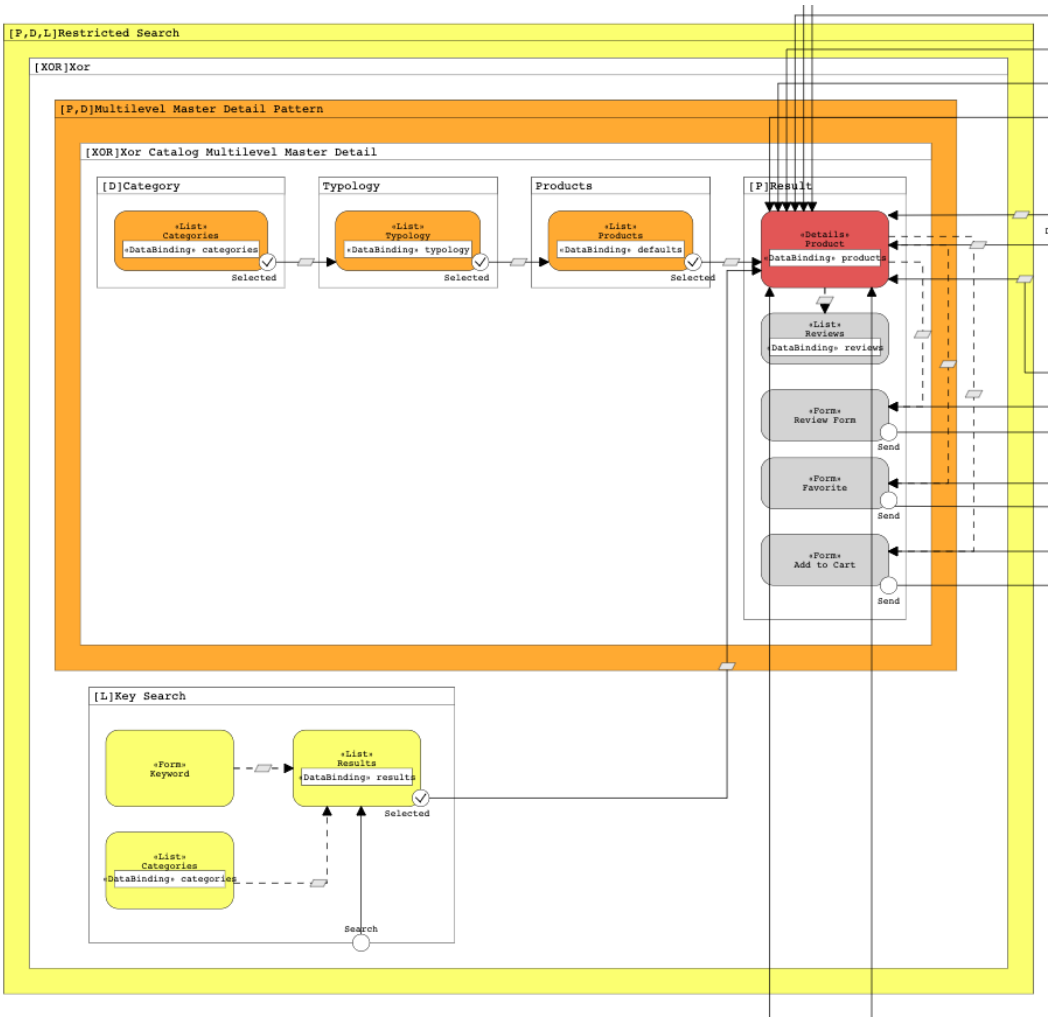


Figure 5.8: The sharing of components among design patterns reduce the risk of duplication.

5.2.4 Command Recognition

The development of the e-commerce model, through the advanced support of voice recognition, has presented numerous advantages in terms of reduction of the realization efforts. However, it must be recognized that voice commands may be subject to interpretation errors. The problem is mainly due to the neural network of Alexa that processes the sentences uttered trying to extract from them the information necessary to understand which command to execute, as well as the values of the parameters required. Regarding the recognition of the commands, no particular problems were found: the orders given were regularly understood. As regards the comprehension of the parameter values, on the other hand, a non-negligible error rate was recorded, above all in the renaming operations of the elements. The reason is attributable to the fact that, during the realization of the Model Creator skill, for each intent, it is required to define the types of each parameter that constitutes it. Therefore, for each parameter it is possible to assign a default type (within a vast list made available by Amazon developers) or create a custom type to which examples must be provided, in orderd to train the neural network. The problem is that the names that can be assigned to the elements of a model are not known a priori and can vary a lot, depending on the context to which the model refers. It follows that the examples that can be provided to train the network, although numerous, are however limited and sometimes may not be sufficient for the network to understand the correct value at run time.

5.3 Final Considerations

The implementation of a voice recognition tool in support of the realization of the models, allows us to formulate considerations that concern aspects that are collateral to the entire application development process.

5.3.1 Customer Relationships

The guided assistance in the generation of models allows reducing the existing gap between customers and developers. Indeed, the voice assistant is able to offer modeling support even to those who do not have technical skills. The questions that the voice assistant addresses, during the interaction process, are related to the requirements and specifications of the final applications and have nothing to do with the modeling language used. Therefore, they are questions that can be answered by users who do not know the editor or IFML. In other words, the system offers a mapping tool from a domain of unqualified people to a domain of specialized people. The current capabilities of the tool, in this sense, are still small, but proves that it is possible to start from functional requirements to develop a model that can potentially be delivered to a developer, in order to have a formal definition of what the client requires (as opposed to a textual document, often subject to interpretation, ambiguity, and misunderstanding).

5.3.2 Accessibility

The integration of a system for the recognition of voice commands within the framework allows people with injuries or motor disabilities to build models with

all the expressive power provided by the editor, with the use of the mouse and the keyboard). This represents a great improvement in terms of accessibility and constitutes an advanced feature for a tool oriented to the development of applications.

Chapter 6

Conclusions and Future Work

In this thesis work, we presented an approach of pattern-based generation of models, in the context of Model Driven Development. The purpose of the work has been to explain how the usage of tools in support of the automatic generation of design patterns can bring improvements (in terms of time and quality of the generated models), which propagated in all the phases of the development cycle, from the design until the complete realization, for the benefit of both developers and end users. All efforts have been directed to the promotion of this idea and the development of all the software systems necessary to ensure that this could be proved.

We showed the potentialities of a tool realized in our reference platform, IFML-Edit.org, able to generate design patterns and insert them within a model, automatically. Secondly, we exposed an algorithm based on a tagging principle, able to detect design patterns inside a model and make changes at the level of pattern, rather than at the basic level of element. Finally, we proposed the use of the artificial intelligence made available by Amazon Alexa in order to support the modeling process inside the editor, through a voice assistant provided with advanced functionalities.

Summing up, in numbers, we worked at the definition of 11 design patterns for the automatic design pattern generator, the development of 27 commands for the building of models through the advance voice assistant, and the realization of 4 interaction flows, for the automatic construction of complete models, through the guided voice assistant.

The results obtained from the examination of the proposed case study demonstrate that it is possible to reduce modeling efforts beyond 70%, increasing at the same time the quality of the final models.

6.1 Future Work

The proposed work can be evolved in different ways. From the point of view of the automatic design patterns generator, we suggest extending the number of patterns that can currently be created by the tool. Furthermore, we promote the definition of metrics that allow providing a percentage of complement of the patterns and the implementation of autocomplete commands, whenever a developer inserts and connects new elements in ways that lead to think that he intends to create (even manually, i.e. without the aid of the automatic generator) a pattern within the model. Looking at the support offered by the vocal assistant, instead, we are convinced that it can represent a tool with enormous potential. Therefore, we point out the following

possible enhancements:

- The numerical increase of the types of complete models that can be developed through guided assistance, guaranteeing its usage for the realization of a wider range of specifications and applications.
- The development of an algorithm or a system of artificial intelligence that consents to build complete models in a dynamic and programmatic way, according to the needs expressed by the developer and without having to refer to preloaded models (which constitute a non-scalable solution).
- Provide greater support for model generation through the advanced voice assistance. Currently, the voice assistant replaces manual work but does not offer advises on future moves, reports of errors and automatic completion of started operations. This would allow the developer to work with an assistant that guarantees greater security and correctness, further reducing modeling time.

Finally, with an interest at improving the performance evaluation, we suggest the definition of an algorithm that automatically and in real time calculates the number and the typology of operations performed, the number of elements present in the model and the time elapsed since the beginning of the modeling process until its complete realization. This would make it possible to objectively test the development processes, with and without the tools implemented to improve performance. In order to evaluate the quality of the applications generated starting from the models, on the other hand, we consider appropriate to define tasks that end users must complete in short times and in a number of pre-established steps (also in this case, a statistical comparison could be performed on the results obtained on applications that make use the designed tools at modeling time versus the same applications implemented from models that does not make use these support tools during the generation process).

Bibliography

[1] Novia Indriaty Admodisastro and Sellappan Palaniappan. "A code generator tool for the gamma design patterns". In: *Malaysian Journal of Computer Science* 15.2 (2002), pp. 94–101.

[2] Ellen Agerbo and Aino Cornils. "How to preserve the benefits of design patterns". In: *ACM SIGPLAN Notices*. Vol. 33. 10. ACM. 1998, pp. 134–143.

[3] Icek Ajzen and Thomas J Madden. "Prediction of goal-directed behavior: Attitudes, intentions, and perceived behavioral control". In: *Journal of experimental social psychology* 22.5 (1986), pp. 453–474.

[4] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[5] Stephen C Arnold, Leo Mark, and John Goldthwaite. "Programming by voice, VocalProgramming". In: *Proceedings of the fourth international ACM conference on Assistive technologies*. ACM. 2000, pp. 149–155.

[6] Colin Atkinson and Thomas Kuhne. "Model-driven development: a metamodeling foundation". In: *IEEE software* 20.5 (2003), pp. 36–41.

[7] Albert Bandura. "Self-efficacy: toward a unifying theory of behavioral change." In: *Psychological review* 84.2 (1977), p. 191.

[8] Ian Bayley and Hong Zhu. "Formalising design patterns in predicate logic". In: *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*. IEEE. 2007, pp. 25–36.

[9] Ian Bayley and Hong Zhu. "Specifying behavioural features of design patterns in first order logic". In: *2008 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE. 2008, pp. 203–210.

[10] Kent Beck and Ralph Johnson. "Patterns generate architectures". In: *European Conference on Object-Oriented Programming*. Springer. 1994, pp. 139–149.

[11] Carlo Bernaschina, Sara Comai, and Piero Fraternali. "Formal semantics of OMG's Interaction Flow Modeling Language (IFML) for mobile and rich-client application model driven development". In: *Journal of Systems and Software* 137 (2018), pp. 239–260.

[12] Carlo Bernaschina, Sara Comai, and Piero Fraternali. "IFMLEdit. org: model driven rapid prototyping of mobile apps". In: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press. 2017, pp. 207–208.

[13] Alex Blewitt, Alan Bundy, and Ian Stark. "Automatic verification of design patterns in Java". In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM. 2005, pp. 224–232.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. "Model-driven software engineering in practice". In: *Synthesis Lectures on Software Engineering* 1.1 (2012), pp. 1–182.

[15] Marco Brambilla and Piero Fraternali. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[16] Frank J. Budinsky et al. "Automatic code generation from design patterns". In: *IBM systems Journal* 35.2 (1996), pp. 151–171.

[17] James O Coplien and Douglas C Schmidt. *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co., 1995.

[18] H Russel Douglas. *Method and apparatus for editing documents through voice recognition*. US Patent 5,875,429. 1999.

[19] Emanuele Falzone. "An approach for integrating code generation and manual development with conflict resolution". Accessed: 2019-02-19. master thesis. Politecnico di Milano.

[20] Emanuele Falzone and Carlo Bernaschina. "Intelligent Code Generation for Model Driven Web Development". In: *International Conference on Web Engineering*. Springer. 2018, pp. 5–13.

[21] Emanuele Falzone and Carlo Bernaschina. "Model Based Rapid Prototyping and Evolution of Web Application". In: *International Conference on Web Engineering*. Springer. 2018, pp. 496–500.

[22] Martin Fishbein and Icek Ajzen. "Belief, attitude, intention, and behavior: An introduction to theory and research". In: (1977).

[23] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[24] Gaston Godin et al. "Healthcare professionals' intentions and behaviours: A systematic review of studies based on social cognitive theories". In: *Implementation Science* 3.1 (2008), p. 36.

[25] Yann-Gaël Guéhéneuc and Rabih Mustapha. "A simple recommender system for design patterns". In: *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories* (2007).

[26] Brent Hailpern and Peri Tarr. "Model-driven development: The good, the bad, and the ugly". In: *IBM systems journal* 45.3 (2006), pp. 451–461.

[27] J. Q. Huang. "A Designer's Assistant Tool". PhD thesis. Ph.D. Thesis, 1996.

[28] Tomas Hustak and Ondrej Krejcar. "Principles of Usability in Human-Computer Interaction". In: *Advanced Multimedia and Ubiquitous Engineering*. Springer, 2016, pp. 51–57.

[29] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[30] Foutse Khomh and Yann-Gaël Guéhéneuc. "Design patterns impact on software quality: Where are the theories?" In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 15–25.

[31] Foutse Khomh and Yann-Gael Gueheneuce. "Do design patterns impact software quality positively?" In: *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE. 2008, pp. 274–278.

[32] Erkki Kilpinen. “Human beings as creatures of habit”. In: (2012).

[33] Danny B Lange and Yuichi Nakamura. “Interactive visualization of design patterns can help in framework understanding”. In: *ACM Sigplan Notices*. Vol. 30. 10. ACM. 1995, pp. 342–357.

[34] David Mendoza and M Hall. “SCUPE (Santa Clara University Pattern Editor)”. MA thesis. Master’s Thesis, 1998.

[35] Jörg Niere et al. “Towards pattern-based design recovery”. In: *Proceedings of the 24th international conference on Software engineering*. ACM. 2002, pp. 338–348.

[36] OMG. *Interaction Flow Modeling Language (IFML), version 1.0*. http://www.omg.org/spec/IFML/1.0. 2015.

[37] *Pattern*. https://it.wikipedia.org/wiki/Pattern. Accessed: 2019-02-18.

[38] D Janaki Ram, KN Raman, and KN Guruprasad. “A pattern oriented technique for software design”. In: *ACM SIGSOFT Software Engineering Notes* 22.4 (1997), pp. 70–73.

[39] Dirk Riehle and Heinz Züllighoven. “Understanding and using patterns in software development”. In: *Theory and practice of object systems* 2.1 (1996), pp. 3–13.

[40] Bran Selic. “The pragmatics of model-driven development”. In: *IEEE software* 20.5 (2003), pp. 19–25.

[41] Harry C Triandis. “Values, attitudes, and interpersonal behavior.” In: *Nebraska symposium on motivation*. University of Nebraska Press. 1979.

[42] Frank Truyen. “The fast guide to model driven architecture the basics of model driven architecture”. In: *Cephas Consulting Corp* (2006).

[43] *Understand Custom Skills*. https://developer.amazon.com/docs/custom-skills/understanding-custom-skills.html. Accessed: 2019-02-24.

[44] *Unified modeling language (UML)*. http://www.uml.org. Accessed: 2019-02-20.

[45] Sven Wenzel and Udo Kelter. “Model-driven design pattern detection using difference calculation”. In: *Workshop on Pattern Detection for Reverse Engineering*. 2006.

[46] Roel Wuyts. “Declarative reasoning about the structure of object-oriented systems”. In: *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No. 98EX176)*. IEEE. 1998, pp. 112–124.

[47] Hong Zhu et al. “Tool support for design pattern recognition at model level”. In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 1. IEEE. 2009, pp. 228–233.

Appendices

Appendix A

Design Patterns

The appendix shows all the design pattern that it is possible to implement in the developed automatic design pattern generator. For each design pattern is reported a representative image of the corresponding model and brief description of its structure and the problem that solve.
All the presented pattern copy and take inspiration from the work by M. Brambilla and P.Fraternali in [15]

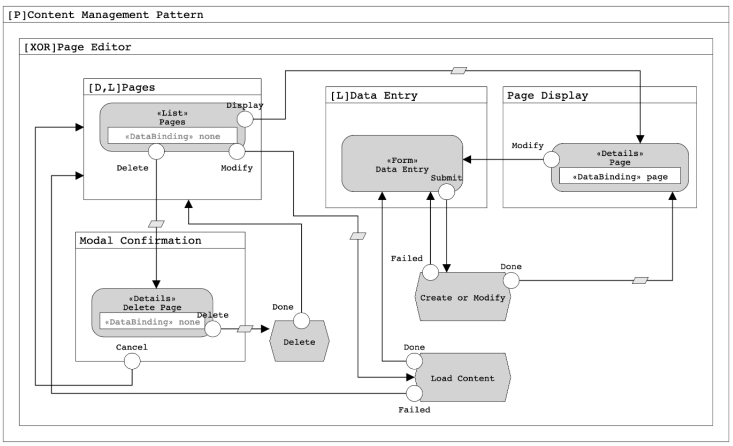
A.1 Content Management Patterns

A.1.1 Content Management

Description

The Content Management pattern address the management of the entire life cycle of an object, including the common operations of creation, modification and deletion of the object itself.

Model



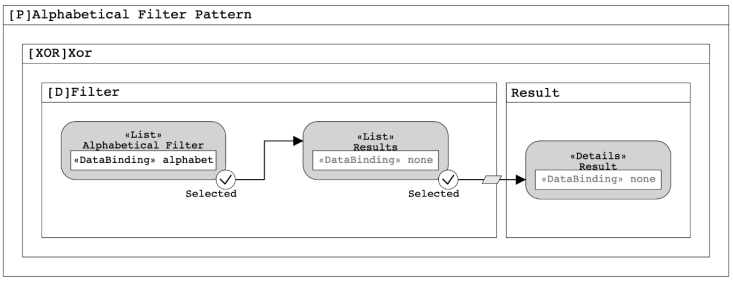
A.2 Content Navigation Patterns

A.2.1 Alphabetical Filter

Description

When the objects to be accessed are numerous but possess an essential attribute that allows them to be accessed alphabetically, a useful pattern provides an alphabetic filter to partition the collection into chunks.

Model

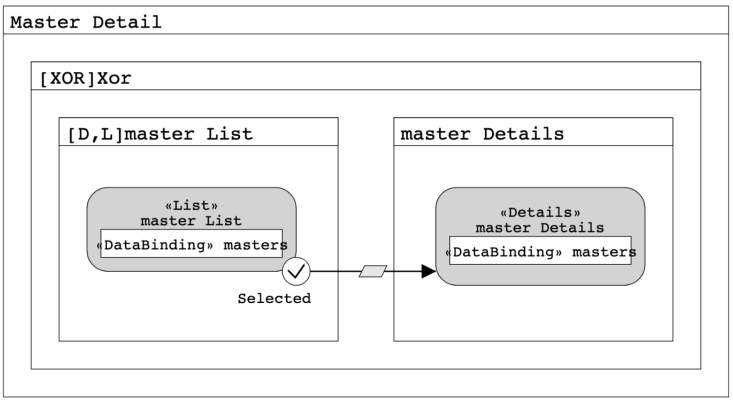


A.2.2 Master Detail

Description

The Master Detail pattern denotes the most straightforward data access pattern. A list ViewComponent is used to present some instances (the so-called master list), and a selection Event permits the user to access the details of one instance at a time.

Model

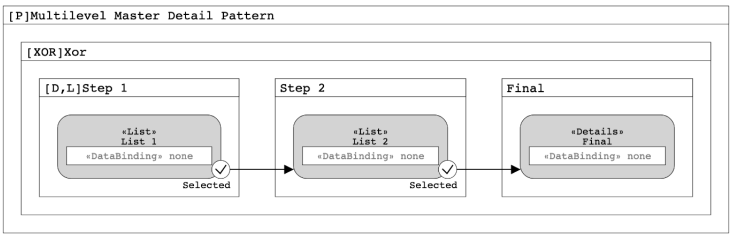


A.2.3 Multilevel Master Detail

Description

Sometimes called "cascaded index", it consists of a sequence of List ViewComponents

defined over distinct classes, such that each List specifies a change of focus from one object (selected from the index) to the set of objects related to it via an association role. In the end, a single object is shown in a Details ViewComponent. A typical usage of the pattern exploits one or more data access classes to build a navigation path to the instances of a core class. Model



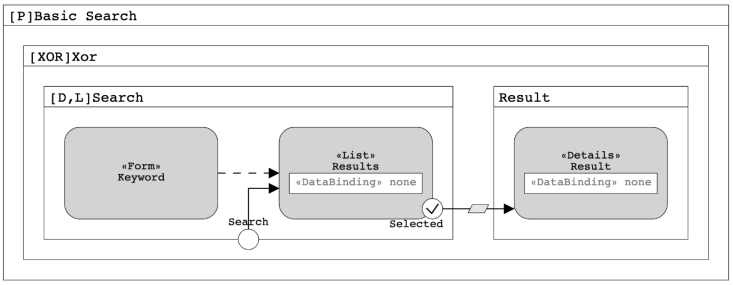
A.3 Content Search Patterns

A.3.1 Basic Search

Description

In the Basic Search pattern, a Form ViewComponent with one simple field is used to input a search key. This key is used as the value of a parameter in the ConditionalExpression of a List ViewComponent that displays all the instances of a class that contains the keyword.

Model

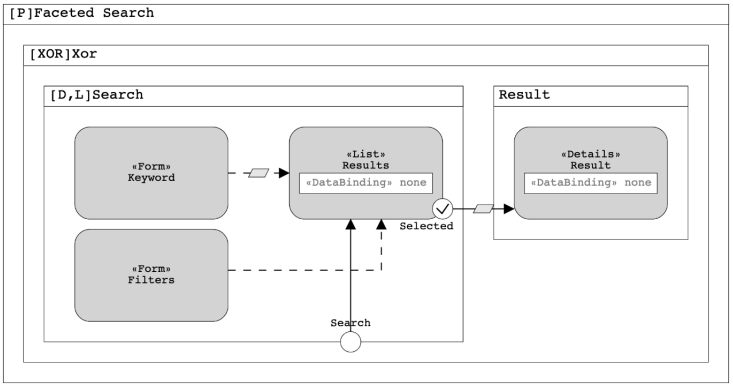


A.3.2 Faceted Search

Description

Faceted Search is a modality of information retrieval particularly well suited to multidimensional structured data. It is used to allow the progressive refinement of the search results by restricting the objects that match the query based on their properties, called facets. By selecting one or more values of some of the facets, the result set is narrowed down to only those objects that possess the selected values.

Model

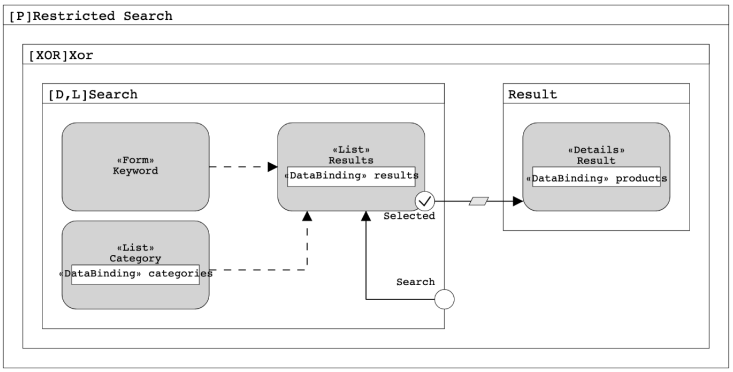


A.3.3 Restricted Search

Description

Search over large collections of objects can be made more efficient by restricting the focus to specific subcollections. This can be performed through this mixed pattern that exploits both content search and access categories.

Model



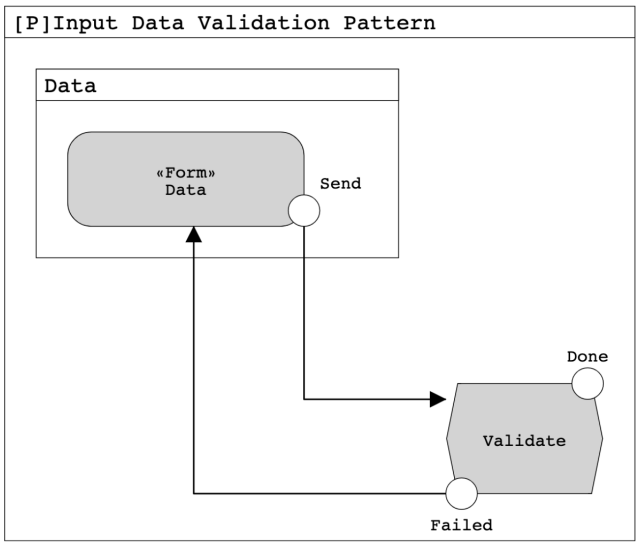
A.4 Data Entry Patterns

A.4.1 Input Data Validation

Description

The Input Data Validation pattern ensures that the input provided by the user meets the application requirements.

Model

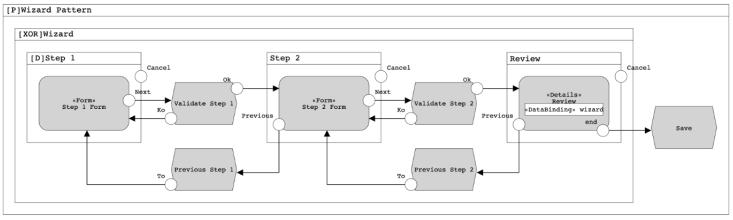


A.4.2 Wizard

Description

The Wizard design pattern supports the partition of a data entry procedure into logical steps that must be followed in a predetermined sequence. Depending on the step reached, the user can move forward or backward without losing the partial selections made up to that point.

Model



A.5 Identification Authorization Patterns

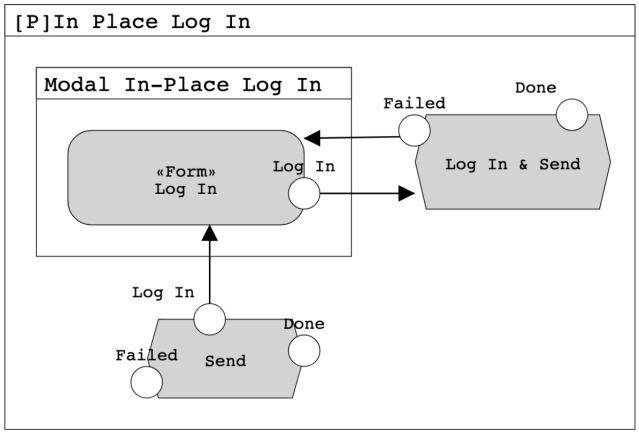
A.5.1 In-Place Log In

Description

The In-Place Log In pattern, typical of the web applications, occurs when a user who is not currently authenticated in the application wants to perform an action that requires identification. When the user attempts to trigger the action, he must be warned of the need to sign in first and be routed to the Log In form. When the user

has successfully signed in, he must be redirected to the interface element from which he requested the initial action. When handling the submission of information, any data entered prior to the Log In procedure must also be preserved.

Model

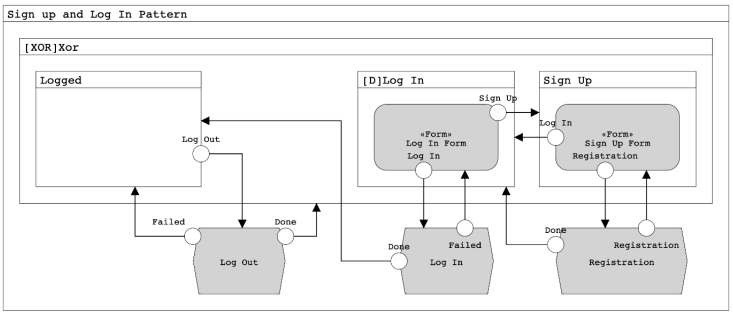


A.5.2 Sign Up and Log In

Description

This pattern addresses the registration, identification, and authorization of the user within the application. After registering on the platform, the user enters credentials using a Form in a public access ViewContainer, and such input is verified against the contest of an identity repository. Upon success, a *Normal Termination* ActionEvent is raised by the Log In Action, the user is authenticated, and this information is preserved in the Context. Upon failure, the Exceptional Termination ActionEvent is raised, which can be trapped by the application to give the user an appropriate warning message. The information about the user's authenticated identity preserved in the Context can be cleared by the initiative of the user by means of a *Log Out*Action. The typical pattern comprises an event that triggers the Action, which normally terminates without exceptions. After the *Log Out* Action is completed, the user is shown the same source ViewContainer, but as a side effect of the logout process, the identification information in the Context is no longer defined and the access to all the private information of the application is denied.

Model



Appendix B

Model Creator Skill Commands

The appendix shows and describes all the commands that it is possible to express in the Model Creator Skill, in order to perform actions in IFMLEdit.org.

B.1 Invocation

This section shows how to:

- Invoke and open the *Model Creator* skill through a voice command (*Open Skill* intent)
- Demand for the creation of a model (*Create Model* intent)
- Choose the type of support you want to obtain from the voice assistant (*Model Type* intent)

B.1.1 Open Skill

Description

Demand the voice assistant to open the skill and create a communication with the IFML framework.

Command

open Model Creator

B.1.2 Create Model

Description

Express the desire to build a model. The execution of the command starts an interaction between the user and the voice assistant.

Command

I want to build a model

Alternative commands

I want to build an application
I want to create a model
Help me to build a model
Help me to build an application

B.1.3 Model Type

Description

After the user requires to build a model with the *Create Model* command, the voice assistant ask the user what type of model he wants to generate.
The types supported by the application are:

- **Guided**: this option assumes the user is a beginner developer who does not want to create a model with special features or the design of the model does not represent a relevant aspect in the building process of an application. According to this choice, the voice assistant turns the interaction to specific questions in order to understand what is the scope of the application and which features it must present. The voice assistant acquires all the information and, at the end of the interaction, it demands the framework to load the model which best fits the user needs (inside a collection of predefined templates).
- **Advanced**: this alternative option imply the user is a skilled developer that desire to build a sophisticated model with specific features. In this case, the voice assistant listens for commands representing the will to create, delete, connect or change the settings of an element within the framework. In other words, the building process of a model is entirely under the control of the user that can execute operations simply through the voice, smartly and efficiently, without losing time dragging and dropping element inside the board with the mouse, changing the parameters with the keyboard, *et alia*.

Voice Assistant request

Do you need to be guided or build an advanced model?

Answer

I want to build a/an {type} model

Alternative answers

I need to build a/an {type} model
I want to build a model following specifics *I want to build a model independently*[1]
I want to be guided
I need to be guided

Parameters

- *type*
  the type of model the developer wants to create. Admitted values are: *guided* and *advanced* (or synonyms).[1]

B.2 Guided Model

Whenever a user requires to be guided in the building of a new model, the voice assistant moves the interaction through a different modality (also called *multi-turn dialog*) where the role of the two interactants is reversed. In other terms, the voice assistant takes control of the dialog and ask specific questions to the user. Each answer leads to the definition of new questions until the voice assistant obtains all the necessary information to build the correspondent model. The information is sent to the framework that receives them and selects and load the model that best fits the user needs.

B.2.1 Application Purpose

Description

The first information required by the voice assistant during the guided interaction is the purpose for which the model of the application must be built. The developer could interpret this question as a conditional statement where, on the base of the answer, the voice assistant change the formulation of the next questions, addressing the dialogue towards the ultimate intent of the user. In particular, the voice assistant can offer support in the building of the model corresponding to three application typologies:

- Show and sell products application: this typology refers to the creation of a standard e-commerce application where the final users can navigate and see a list of products, add items to the cart and buy them.
- Share contents application: this alternative option may concern two different subcategories of applications: social networks and blogs (or communities). In this case, the voice assistant requests further information, in order to discern the real intent of the developer.
- Collaboration application: this additional typology leads to the realization of a sort of crowdsourcing platform where people can ask other people to perform tasks.

Voice Assistant request

The primary purpose of the application is to show and sell products, share content or ask people to collaborate on some project?

[1] for example, *specifics* and *independently* are considered as synonym for the key term *advanced*. The voice assistant is governed by a neural network able to take in input sample phrases with terms and synonyms and interpret new sentences with the same meanings, bringing it back to the same command or choice.

Answer

The purpose of the application is to {purpose}

Alternative answers

I need to {purpose}
I want to build a {purpose} application

Parameters

- *purpose*
  the purpose of the final application. Admitted values are: *show and sell products*, *share contents* and *collaborate* (or synonyms).

B.2.2 Show and sell products

Determined the willingness of the developer, the interaction moves towards specifics questions related to which components the e-commerce application must implement.

B.2.2.1 Products Classification

Description

The voice assistant requires to know the typology of products the final user intends to insert inside the application. In particular, the voice assistant expects the products may be:

- Homogeneous: the products will be similar and they will change only for a few negligible properties. We can say they will not be discernable in their primary purpose, features, and functionality.
- Heterogeneous: the products will be various and characterized by not negligible different features. We can say they will be discernable in their primary purpose, features, and functionality and consequently they may be divided into category and subcategories.

The answer reported by the user determines what type of organizational and search patterns will be inserted in the model, related to the items that will populate the catalog in the final application. According to the design of the intent, *homogeneous* products will be organized through the *master detail* pattern and the correlated search system will be implemented using the simple *basic search* pattern. Conversely, *heterogeneous* items will be arranged in categories and subcategories, through the *multilevel master detail* pattern and the associated search system will be implemented using the more sophisticated *restricted search* pattern.

Voice Assistant request

The products you intend to sell are homogeneous or require to be divided into categories and sub-categories?

Answer

They are {typology}

Alternative answers

They can be divided in categories and subcategories
They will be homogeneous

Parameters

- *typology*
  the typology of products that will populate the application. Admitted values are: *homogeneous* and *heterogeneous* (or synonyms).

B.2.2.2 Reviews Policy

Description

Large distribution companies that sell products around the world and let third-party companies sell products in their portal, usually adopt a review policy according to which any final user can give feedback about the bought products. This feature represents a significant indicator for the new potential buyer respect to the reliability of the seller of a product and the quality of the same. On the contrary, small companies that want to open their own online sales portal may want to protect themselves by not exposing their products to the opinions of the users. According to the policy adopted in the first case, the *reviews content management* pattern will be implemented in the final model, while in the second case the application will not offer the opportunity to comment and review products.

Voice Assistant request

Do you want to allow or deny reviews of the products?

Answer

{commentsPolicy} reviews

Alternative answers

I want to {commentsPolicy} reviews
Yes allow
No deny

Parameters

- *commentsPolicy*
  the policy the developer wants to adopt about the comments and the reviews of the products. Admitted values are: *allow* and *deny* (or synonyms).

B.2.2.3 Favorite Policy

Description

Similarly, it is at the discretion of the developer to choose if the final users can save the preferred products in a sort of wish list. If the developer *allows* this policy, the final model will implement the *favorite content management* pattern, otherwise not.

Voice Assistant request

It would be nice if users could save their favorite products on a wish list. What do you think about it?

Answer

$\{favoritePolicy\}$ *wish list*

Alternative answers

Yes allow
No deny
I think it's a great idea[2]

Parameters

- $favoritePolicy$
  the policy the developer wants to adopt about the wish list. Admitted values are: *allow* and *deny* (or synonyms).

B.2.2.4 Payment Policy

Description

Finally, the voice assistant suggests the use of the *wizard* pattern in order to manage the payment process, progressively acquiring all the information (about the delivery address, the billing address, the credit card, *et alia*), needed to conclude an order. The developer can decide whether to *accept* or *reject* the proposal.

Voice Assistant request

The payment procedure is managed through a wizard pattern that progressively requires information to the customer. It is good for you or do you want to create your personal procedure independently?

Answer

I prefer to use the default payment procedure
I want to create my personal procedure

Alternative answers

It is good for me[2]

[2] This answer is intended as an opening towards the proposal promoted by the voice assistant

B.2.3 Share contents

Description

As previously mentioned, the choice to build an application with the primary purpose of share contents is rather ambiguous. More in detail, the type of content referred to could allude to posts (containing images, text, links) or articles of various kinds. Consequently, the final application the developer intends to implement could be a social network rather than a blog or a sort of community. The voice assistant tries to dissolve any doubt by asking the user what kind of content is intended to be shared in the final application. Based on the answer, the voice assistant will guide the developer in the building of the right type of model, with specific questions. **Voice**

Assistant request

What type of content users can share, posts or articles?

Answer

They can share {*type*}

Alternative answers

Users will share {*type*}

Parameters

- *type*
  the type of contents will be shared in the final application. Admitted values are: *posts* and *articles* (or synonyms).

B.2.3.1 Social Network

If the type of contents the users will share in the final application are post, the voice assistant turns the interaction towards the building of a model that best represents a sort of social network.

B.2.3.1.1 Comments Policy

Description

Usually, inside a social network users can react to a post of another user commenting or criticizing the shared content. However, nothing forbids the possibility to create an alternative social network where the users can react to a post in a different and limited way (for example, emoticons which represent filtered emotions that cannot harm the publisher). It is a discretion of the developer to choose which policy to adopt for his model (consequently, the final model will implement or not the *comment content management* pattern).

Voice Assistant request

Can user comment posts or they can only visualize them?

Answer

{commentsPolicy} comments

Alternative answers

I want to {commentsPolicy} comments
Yes allow
No deny

Parameters

- *commentsPolicy*
  the policy the developer wants to adopt about the comments of the posts. Admitted values are: *allow* and *deny* (or synonyms).

B.2.3.1.2 Likes Policy

Description

Another typical feature of the social networks is the possibility to express a fast reaction to a post, through the so-called like functionality. The developer can adopt this traditional feature also in his application (in which case the final model will implement the *like content management* pattern) or reject it.

Voice Assistant request

It would be nice if users could express consent to a post through a like functionality.
What do you think about it?

Answer

I think it is a {likesPolicy} idea

Alternative answers

Yes allow
No deny

Parameters

- *likesPolicy*
  the policy the developer wants to adopt about the like functionality. Admitted values are: *good* and *bad* (or synonyms).

B.2.3.1.3 Relationships Policy

Description

Finally, in traditional social networks, users can establish relations, normally called friendships. The voice assistant asks the developer if its model must present the same functionality or he wants to generate an application where does not exists the concept of relationship. In the first case, the model will implement the *friends content management* pattern and the *basic search* pattern in order to find actual or new potential friends.

Voice Assistant request

In traditional social networks users can establish relationships. Is this your case?

Answer

Yes, it is
No it isn't

Alternative answers

It is {*relationPolicy*} *for me*

Parameters

- *relationsPolicy*
  the policy the developer wants to adopt about the relationships among users. Admitted values are: *good* and *bad* (or synonyms).

B.2.3.2 Blog

If the type of contents the users will share in the final application are articles, the voice assistant turns the interaction towards the building of a model that best represents a sort of blog (or community, in a broad sense).

B.2.3.2.1 Articles Organization

Description

The voice assistant requires to know the typology of articles the user will be able to publish inside the application. In particular, the voice assistant expects the articles may be:

- Homogeneous: the articles will refer to the same topic and they will change only for a few negligible properties.
- Heterogeneous: the articles will be various and relatives to different topics. They may be divided into category and subcategories.

The answer reported by the user determines what type of organizational and search patterns will be inserted in the model, related to the type of elements that will populate the final application. According to the design of the intent, *homogeneous* articles will be organized through the *master detail* pattern and the correlated search system will be implemented using the simple *basic search* pattern. Conversely, *heterogeneous* articles will be arranged in categories and subcategories, through the *multilevel master detail* pattern and the associated search system will be implemented using the more sophisticated *restricted search* pattern.

Voice Assistant request

Are the articles relative to the same topic or they can divided in categories and sub-categories?

Answer

They are {typology} model

Alternative answers

They can be divided in categories and subcategories
They are of the same topic[3]

Parameters

- *typology*
  the typology of articles that will populate the application. Admitted values are: *homogeneous* and *heterogeneous* (or synonyms).

B.2.3.2.2 Community Policy

Description

Traditionally, a blog is considered like a personal web site where the textual narration of personal experiences represents the main contents (be they travel, hobbies, passions, *et alia*). In a few words, we can see a blog like a portal where a single user shares contents and the other users can only read and comment what the owner user publishes. The concept of the blog can be extended (in a broad sense) to that of community one at the moment when every user can publish his own articles and comment the articles of others users, open a constructive discussion and share information about a particular topic. The voice assistant, in this case, requires to know if the developer desires that in the final application every user can publish and manage his article or not. In the first case, the model will implement the *personal pages content management* pattern.

Voice Assistant request

Are you interested in creating a sort of community where user can publish articles or users can only visualize and execute operations on published articles?

[3] *same topic* is considered as synonym for the key term *homogeneous*.

Answer

I want a sort of {*communityPolicy*}

Alternative answers

They can publish[4]
They can only visualize[5]

Parameters

- *communityPolicy*
  the choice that determines if final users can only read or they can also publish articles. Admitted values are: *blog* and *community* (or synonyms).

B.2.3.2.3 Favorite Policy

Description

Similarly, it is at the discretion of the developer to choose if the final users can save the preferred articles in a sort of favorite list. If the developer *allows* this policy, the final model will implement the *favorite content management* pattern, otherwise not.

Voice Assistant request

It would be nice if users could save the most beautiful articles on a favorite list. What do you think about it?

Answer

{$favoritePolicy$} *favorite list*[6]

Alternative answers

Yes allow
No deny
I think it's a great idea[2]

Parameters

- $favoritePolicy$
  the policy the developer wants to adopt about the favorite list. Admitted values are: *allow* and *deny* (or synonyms).

[4] *publish* is considered as synonym for the key term *community*.
[5] Similarly, *visualize* is considered as synonym for the key term *blog*.
[6] $favoritePolicy$ is a parameter that can assume the values *allow* or *deny*

B.2.3.2.4 Comments Policy

Description

Finally, the developer is invited to choose if the final application will have to provide the possibility to comment articles, implementing the *comments content management* pattern to manage them.

Voice Assistant request

Do you want to allow or deny comments to the articles?

Answer

{*commentsPolicy*} *comments*

Alternative answers

I want to {*commentsPolicy*} *comments*
Yes allow
No deny

Parameters

- *commentsPolicy*
  the policy the developer wants to adopt about the comments of the articles. Admitted values are: *allow* and *deny* (or synonyms).

B.2.4 Collaboration

The last typology of guided model refers to the generation of applications where people can collaborate in the resolution of problems and tasks.

B.2.4.1 Tasks Policy

Description

In common crowdsourcing platforms, users can publish tasks that others can fulfill. In this terms, users can be divided in publishers (call them *master* users) and executors (call them *worker* users). When a user registers within the application, he must declare what type of role he intends to cover. However, nothing forbids to create a different application where only the owners of the platform can publish tasks. In this particular case, the users can only register as *workers* and perform tasks. Speaking about the corresponding final model, in the first case, the application will present a log in form that will redirect to the *master* interface or the *worker* one, in accordance with the role played by the user, while in the second case, the application will redirect to the only user interface provided for the executors.

Voice Assistant request

Are users limited to perform tasks or some users can publish tasks?

Answer

They can only perform tasks
They can also publish tasks

Alternative answers

the first[7]
the second[8]

B.2.4.2 Tasks Classification

Description

Finally, the voice assistant requires to know the typology of tasks will populate the platform. In particular, the voice assistant expects the tasks may be:

- Homogeneous: the tasks will be similar and they will change only for a few negligible properties. We can say they will not be discernable in their primary purpose, topic, and difficulty.
- Heterogeneous: the tasks will be various and characterized by not negligible different features. We can say they will be discernable in their primary purpose, topic, and difficulty and consequently they may be divided into category and subcategories.

The answer reported by the user determines what type of organizational and search patterns will be inserted in the model, related to the tasks that will populate the platform in the final application. According to the design of the intent, *homogeneous* tasks will be organized through the *master detail* pattern and the correlated search system will be implemented using the simple *basic search* pattern. Conversely, *heterogeneous* tasks will be arranged in categories and subcategories, through the *multilevel master detail* pattern and the associated search system will be implemented using the more sophisticated *restricted search* pattern.

Voice Assistant request

Are tasks relative to the same topic or they can be divided in categories and subcategories?

Answer

They are {typology} model

Alternative answers

They can be divided in categories and subcategories

[7] *the first* is considered as synonym for the key term *perform*.
[8] Similarly, *the second* is considered as synonym for the key term *publish*.

Parameters

- *typology*
  the typology of tasks that will populate the application. Admitted values are: *homogeneous* and *heterogeneous* (or synonyms).

B.3 Advanced Model

Expert developers can use this option in order to build wholly customized models. In this case, the voice assistant listens to new commands that may require the generation or deletion of a new element, the insertion or removal of attributes and parameters, the relocation of elements within others elements, the creation of patterns, *et alia*.
In other words, the framework offers a multi-modal approach to the generation of models: you can interact physically with the platform, using mouse and keyboard or directly speak with your voice to give orders to the voice assistant that will take care of translating each command into concrete actions.

B.3.1 Add Binding

Description

Create a binding between a field or parameter and a filter, result or another field of two different elements connected through a flow element.

Request

Add binding to {*elementName*} {*elementType*} *with input* {*input*} *and output* {*output*}

Alternative requests

Add binding with input {*input*} *and output* {*output*}

Parameters

- [optional] *elementName*
  the name of the flow element to which you want to add the binding.[9][10]
- [optional] *elementType*
  the type of the element to which you want to add the binding.[10] Admitted values are *navigation flow* and *data flow*.
- *input*
  the name of the field, filter or result of the target element, involved in the binding.
- *output*
  the name of the field or parameter of the source element, involved in the binding.

[9] If provided, the *elementType* parameter becomes mandatory.
[10] If omitted, the voice assistant operate on the element selected through the *select command* (see section B.3.19). If the selected element is `undefined` or invalid, the command will not be performed

B.3.2 Add Field

Description

Add a field to the selected View Component. **Request**

Add {$fieldName$} *field of type* {$fieldType$} *to* {$elementName$} {$elementType$}

Alternative requests

Add {$fieldName$} *field of type* {$fieldType$}

Parameters

- $fieldName$
  the name of the field you want to add.
- [optional] $fieldType$
  the type of the field you want to add. Admitted values are: *text*,*textarea*, *password*, *checkbox*, *radio*, *reset*, *hidden*, and *hidden object*. [11]
- [optional] $elementName$
  the name of the element to which you want to add the field.[9] [10]
- [optional] $elementType$
  the type of the element to which you want to add the field. Admitted values are *list*, *details* and *form*.[10]

B.3.3 Add Filter

Description

Add a filter to the selected List View Component.

Request

Add {$fieldName$} *filter to* {$elementName$} *list*

Alternative requests

Add {$fieldName$} *filter*

Parameters

- $filterName$
  the name of the filter you want to add.
- [optional] $elementName$
  the name of the List View Component to which you want to add the filter.[10]

[11] if omitted the default type *text* is assigned

B.3.4 Add Parameter

Description

Add a parameter to the selected Action.

Request

Add {*parameterName*} *parameter to* {*elementName*} *action*

Alternative requests

Add {*parameterName*} *parameter*

Parameters

- *parameterName*
  the name of the parameter you want to add.
- [optional] *elementName*
  the name of the Action element to which you want to add the parameter.[10]

B.3.5 Add Pattern

Description

Add a pattern within the model board.

Request

Add {*pattern*} *pattern*

Alternative requests

Add {*pattern*} *pattern with* {*steps*} *steps*

Parameters

- *pattern*
  the name of the pattern you want to add. Admitted values for the parameter are: *alphabetical filter*, *basic search*, *content management*, *faceted search*, *in-place log in*, *input data validation*, *master detail*, *multilevel master detail*, *restricted search*, *sign up and log in*, *wizard*.
- [optional] *steps*
  the number of steps the pattern must have. This parameter is mandatory only for those dynamic patterns (i.e. *multilevel master detail* and *wizard* pattern) that may be characterized by iterative components whose number is not known a priori and must be provided.

B.3.6 Add Result

Description

Add a result to the selected Action.

Request

Add {*resultName*} *result to* {*elementName*} *action*

Alternative requests

Add {*resultName*} *result*

Parameters

- *resultName*
  the name of the result you want to add.
- [optional] *elementName*
  the name of the Action element to which you want to add the result.[10]

B.3.7 Connect

Description

Generate a flow between the source element and the target element.

Request

Connect {*elementName*} {*elementType*} *to* {*target*} {*targetType*}

Alternative requests

Connect to {*target*} {*targetType*}

Parameters

- [optional] *elementName*
  the name of the source element.[9] [10]
- [optional] *elementType*
  the type of the source element.[10]
- *target*
  the name of the target element.
- *targetType*
  the type of the target element.

B.3.8 Delete

Description

Delete the element and all the embedded children.

Request

Delete {*name*} {*type*}

Alternative requests

Delete element

Parameters

- [optional] *elementName*
  the name of the element to be deleted.[9] [10]
- [optional] *elementType*
  the type of the element to be deleted.[10]

B.3.9 Drag and Drop

Description

Drag and drop the element and all the embedded children, along the direction provided.

Request

Drag and drop {*elementName*} {*elementType*} {*direction*} *for* {*delta*} *pixels*

Alternative requests

Drag and drop {*direction*} *for* {*delta*} *pixels*

Parameters

- [optional] *elementName*
  the name of the element to be dragged.[9] [10]
- [optional] *elementType*
  the type of the element to be dragged.[10]
- *direction*
  the direction along which the drag and drop operation must be performed. Admitted values are: *up*, *down*, *right* and *left*.
- *delta*
  the number of pixels the element must be moved (a multiple of ten).

B.3.10 Generate

Description

Generate a new element (View Container, View Component or Action).

Request

Generate {*elementName*} {*elementType*}

Alternative requests

Add new {*elementName*} {*elementType*}

Parameters

- *elementName*
  the name of the element to be generated.
- *elementType*
  the type of the element to be generated. Admited values are: *view container*, *list*, *details*, *form* and *action*.[12]

B.3.11 Insert

Description

Insert an element (View Container, View Component, Action, Event) inside an existing View Container. If the element to be inserted is an existing element, move the element and all the embedded children within the destination View Container, otherwise generate it. **Request**

Insert {*elementName*} {*elementType*} *inside* {*parent*} *view container* {*position*} *respect to* {*childName*} {*childType*}

Alternative requests

Insert inside {*parent*} *view container*

Parameters

- [optional] *elementName*
  the name of the element to be inserted.[9] [10]
- [optional] *elementType*
  the type of the element to be inserted. Admited values are: *view container*, *list*, *details*, *form*, *action*, *event*.[10]

[12] To generate a *Navigation Flow* or a *Data Flow* use the *connect* command instead (see section B.3.7). To generate an *Event* use the *insert* command instead. B.3.11

- *parent*
  the name of the existing view container within which the element must be inserted.

- [optional] *position*
  the position respect to the *child* element (embedded in the *parent* view container) where the selected (or new) element must be inserted. Admitted values are: *up*, *down*, *right* and *left*.[13]

- [optional] *childName*
  the name of the *child* element (embedded in the *parent* view container), respect to which the selected (or new) element must be positioned.

- [optional] *childType*
  the type of the *child* element (embedded in the *parent* view container), respect to which the selected (or new) element must be positioned.

B.3.12 Move Board

Description

Move the view of the board within which the model is inserted, along the direction provided.

Request

Move {*direction*} *for* {*times*} *times*

Alternative requests

Move {*direction*}

Parameters

- *direction*
  the direction along which the movement must be performed. Admitted values are: *up*, *down*, *right* and *left*.

- [optional] *times*
  the number of times the command must be repeated. Each time the motion is executed, the board moves 300 pixels along the direction provided.

B.3.13 Remove Binding

Description

Remove an existing binding between a field or parameter and a filter, result or another field of two different elements connected through a flow element.

[13] If *position* parameter is provided, *child* and *childType* parameters become mandatory.

Request

Remove binding to {*elementName*} {*elementType*} *with input* {*input*} *and output* {*output*}

Alternative requests

Remove binding with input {*input*} *and output* {*output*}

Parameters

- [optional] *elementName*
  the name of the flow element to which you want to remove the binding. [9] [10]
- [optional] *elementType*
  the type of the element to which you want to remove the binding.[10] Admitted values are *navigation flow* and *data flow*.
- *input*
  the name of the field, filter or result of the target element, involved in the binding.
- *output*
  the name of the field or parameter of the source element, involved in the binding.

B.3.14 Remove Field

Description

Remove an existing field from the selected View Component.

Request

Remove {*fieldName*} *field of type* {*fieldType*} *to* {*elementName*} {*elementType*}

Alternative requests

Remove {*fieldName*} *field of type* {*fieldType*}

Parameters

- *fieldName*
  the name of the field you want to remove.
- [optional] *fieldType*
  the type of the field you want to remove.[11] Admitted values are: *text*, *textarea*, *password*, *checkbox*, *radio*, *reset*, *hidden*, and *hidden object*.
- [optional] *elementName*
  the name of the element to which you want to remove the field.[9] [10]

- [optional] *elementType*
  the type of the element to which you want to add the field.[10] Admitted values are *list*, *details* and *form*.

B.3.15 Remove Filter

Description

Remove an existing filter from the selected List View Component.

Request

Remove {*fieldName*} *filter from* {*elementName*} *list*

Alternative requests

Remove {*fieldName*} *filter*

Parameters

- *filterName*
  the name of the filter you want to remove.
- [optional] *elementName*
  the name of the List View Component to which you want to remove the filter.[10]

B.3.16 Remove Parameter

Description

Remove an existing parameter from to the selected Action.

Request

Remove {*parameterName*} *parameter from* {*elementName*} *action*

Alternative requests

Remove {*parameterName*} *parameter*

Parameters

- *parameterName*
  the name of the parameter you want to remove.
- [optional] *elementName*
  the name of the Action element to which you want to remove the parameter.[10]

B.3.17 Remove Result

Description

Remove an existing result from the selected Action. **Request**

Remove {*resultName*} *result from* {*elementName*} *action*

Alternative requests

Remove {*resultName*} *result*

Parameters

- *resultName*
  the name of the result you want to remove.
- [optional] *elementName*
  the name of the Action element to which you want to remove the result.[10]

B.3.18 Resize

Description

Resize element along the direction provided.

Request

Resize {*elementName*} {*elementType*} {*direction*} *for* {*sign*} {*delta*} *pixels*

Alternative requests

Resize {*direction*} *for* {*sign*} {*delta*} *pixels*

Parameters

- [optional] *elementName*
  the name of the element to be scaled back.[9] [10]
- [optional] *elementType*
  the type of the element to be scaled back.[10]
- [optional] *type*
  its value define the typology of resizing (growth or shrinkage). Admitted values are: *plus* and *minus*.[14]
- *direction*
  the side of the element along which the resize operation must be performed. Admitted values are: *up*, *down*, *right* and *left*.
- *delta*
  the number of pixels the side of the element must be scaled back (a multiple of ten).

[14] if omitted, the default type *plus* is assigned

B.3.19 Select element

Description

Select an existing element respect to which execute the next commands.

Request

Select {*elementName*} {*elementType*}

Parameters

- *elementName*
  the name of the element to be selected.
- *elementType*
  the type of the element to be selected. Admitted values are: *view container*, *list*, *details*, *form*, *action*, *event*, *navigation flow* and *data flow*.

B.3.20 Set Collection

Description

Set the collection name of the selected View Component.

Request

Assign the name {*collectionName*} *to the collection of* {*elementName*} {*elementType*}

Alternative requests

Assign the name {*collectionName*} *to the collection*

Parameters

- *collectionName*
  the name you want to give to the collection of the selected View Component.
- [optional] *elementName*
  the name of the View Component to which you want to set the collection name.[9] [10]
- [optional] *elementType*
  the type of the element to which you want to set the collection name. Admitted values are *list*, and *details*.[10]

B.3.21 Set Name

Description

Set the name (and consequently the id) of the selected Element. **Request**

Assign the name {*name*} *to* {*elementName*} {*elementType*}

Alternative requests

Assign the name {*name*}

Parameters

- *name*
  the name you want to give to the Element.
- [optional] *elementName*
  the name of the View Component to which you want to set the name.[9] [10]
- [optional] *elementType*
  the type of the element to which you want to set the name.[10] Admitted values are: *view container*, *list*, *details*, *form*, *action*, *event*, *navigation flow* and *data flow*.

B.3.22 Set View Container Properties

Description

Set the properties of the selected View Container.

Request

{*operation*} {*property*} *to/from* {*elementName*} *view container*

Alternative requests

{*operation*} {*property*}

Parameters

- *operation*
  the type of operation you want to apply to the property. Admitted values are: *add* and *remove*.
- *property*
  the property you want to "add to" or "remove from" the View Container. Admitted values are: *landmark and default*, *landmark*, *default* and *xor*.
- [optional] *elementName*
  the name of the View Container to which you want to set the properties.[10]

B.3.23 Set Event Type

Description

Set the type of the selected Event. **Request**

Set the type of {*elementName*} *event to* {*type*}

Alternative requests

Set type to {*type*}

Parameters

- [optional] *elementName*
  the name of the Event to which you want to set the type.[10]
- *type*
  the type you want to assign to the selected Event. Admitted values are: *user*, *selection* and *system*.

B.3.24 Zoom Board

Description

Zoom the view of the board within which the model is inserted.

Request

{*zoom*}

Alternative requests

{*zoom*} *for* {*times*} *times*

Parameters

- *zoom*
  the type of zoom you want to apply. Admitted values are: *zoom in* and *zoom out*.
- [optional] *times*
  the number of times the command must be repeated. Each time the zoom is executed, the board zoom in or out for 50 pixels.

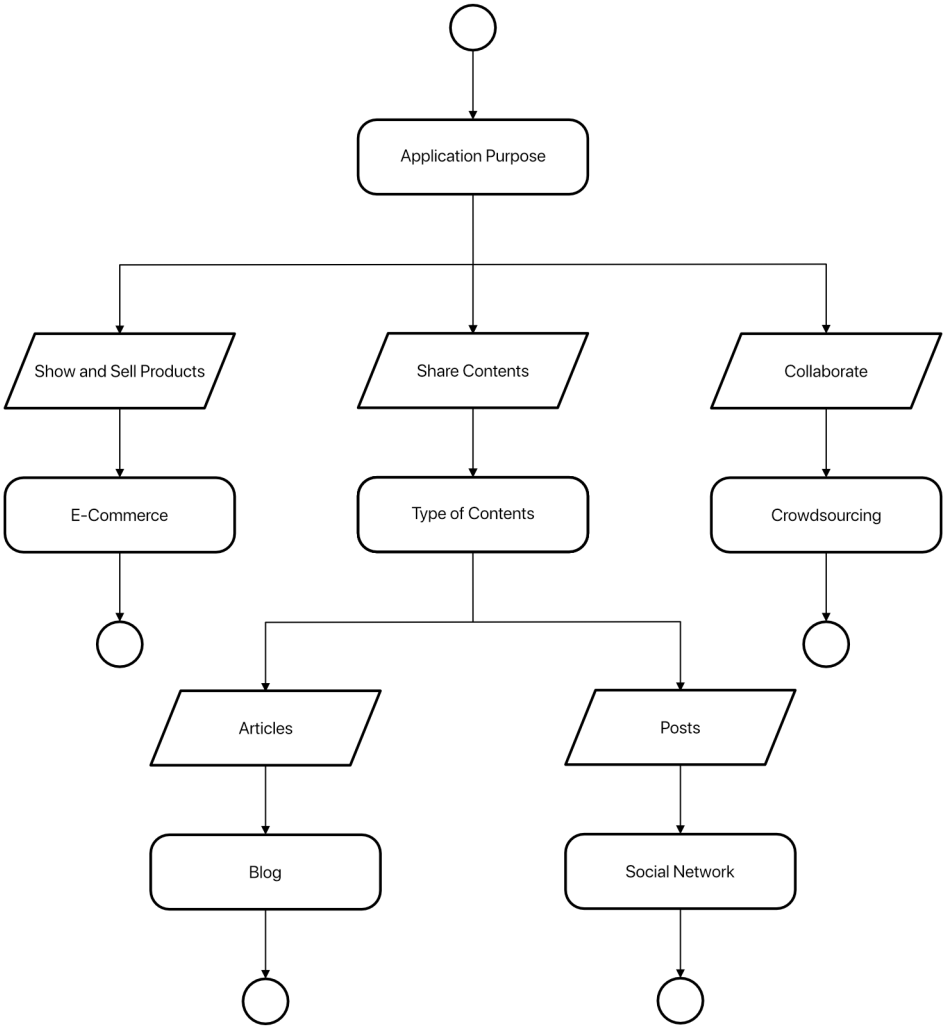


Figure B.1: Guided Model Flowchart

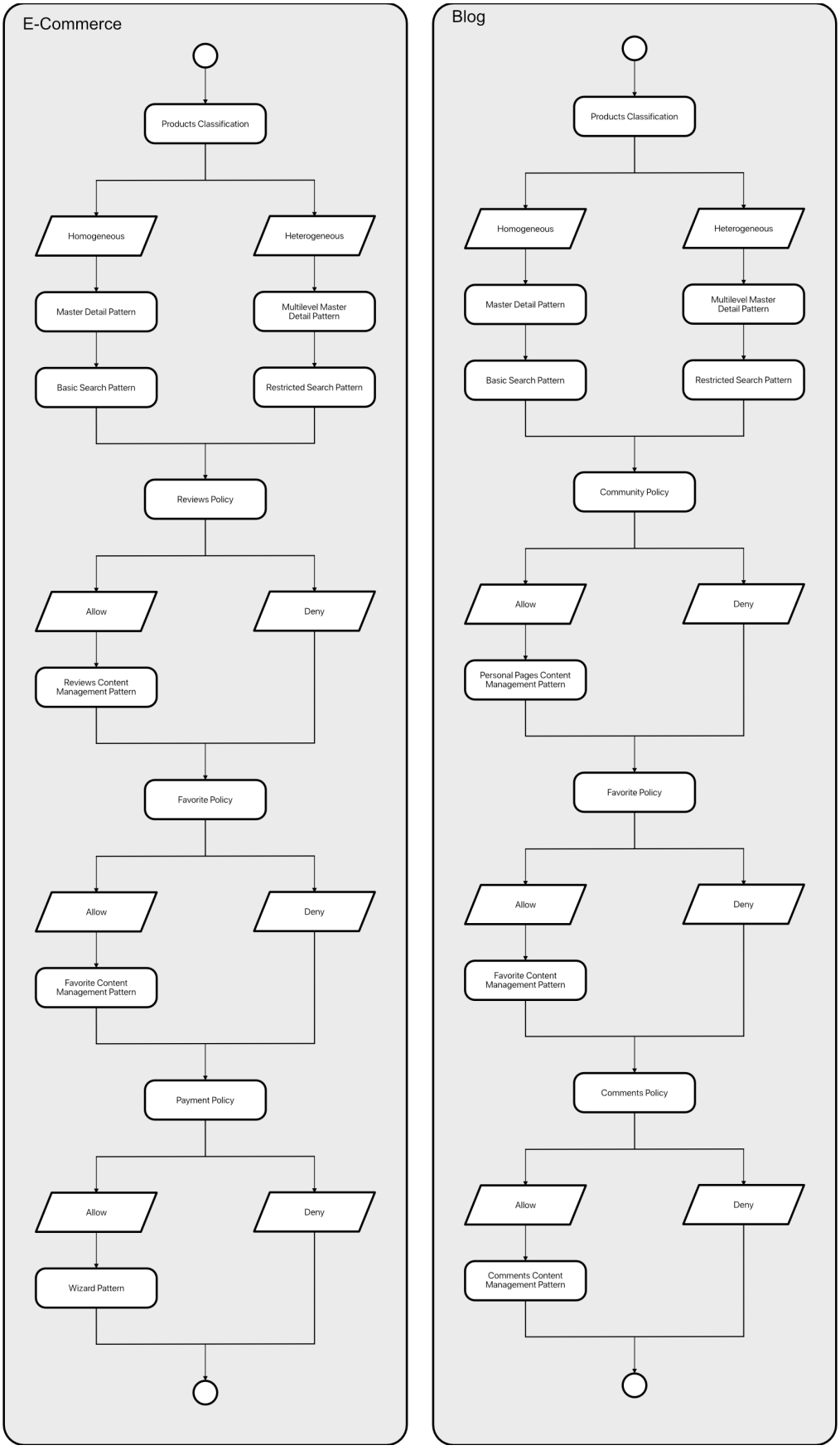


Figure B.2: E-Commerce and Blog model generation flowcharts

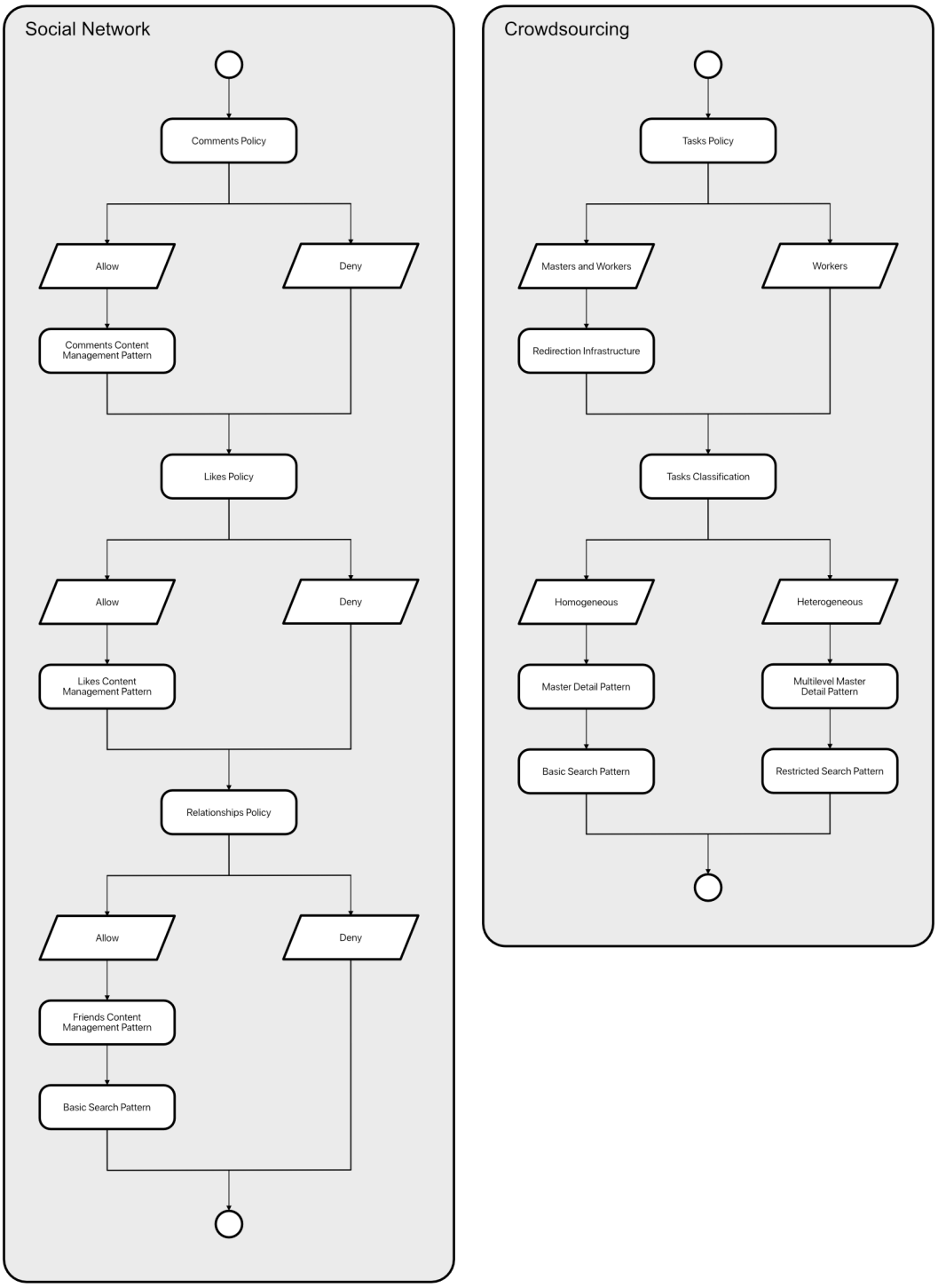


Figure B.3: Social Network and Crowdsourcing model generation flowcharts

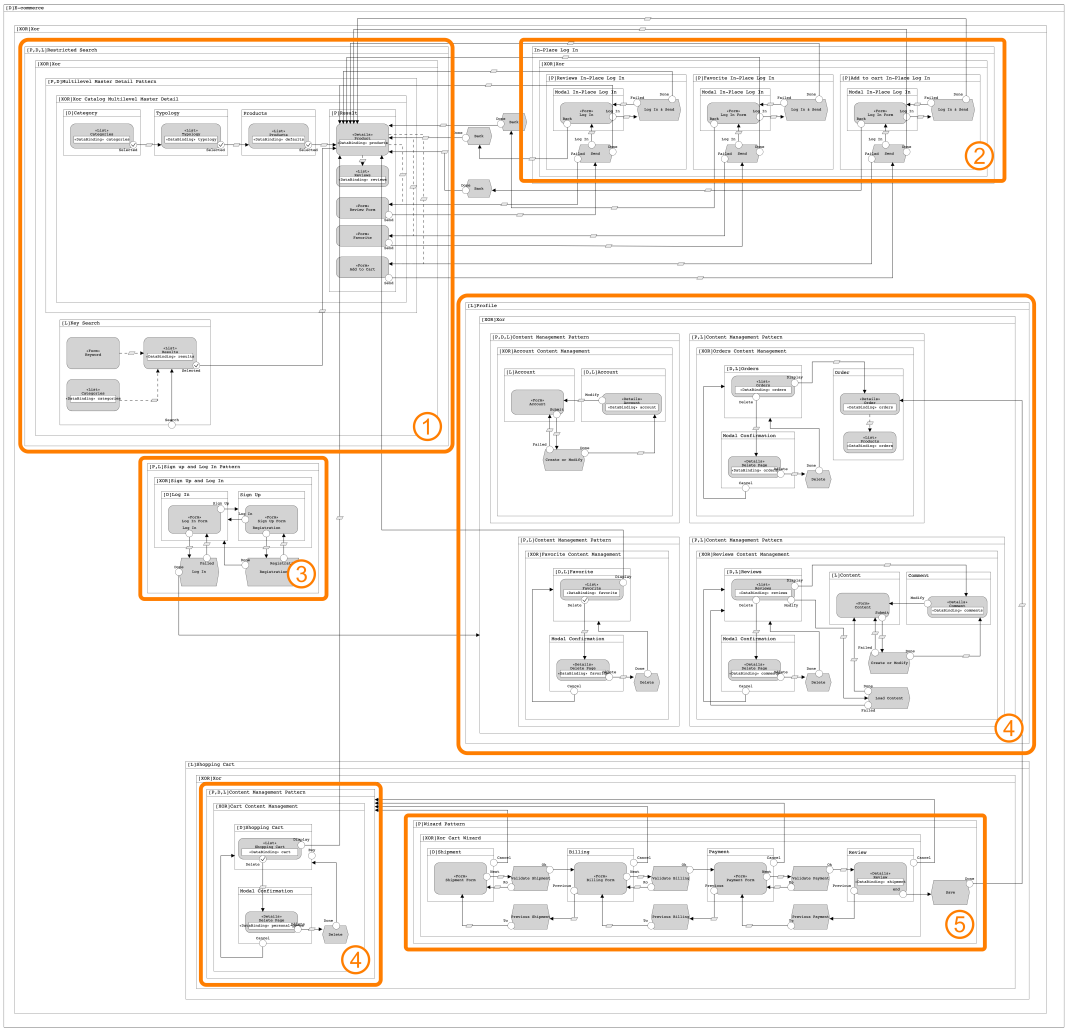


Figure B.4: E-commerce Application Model Example

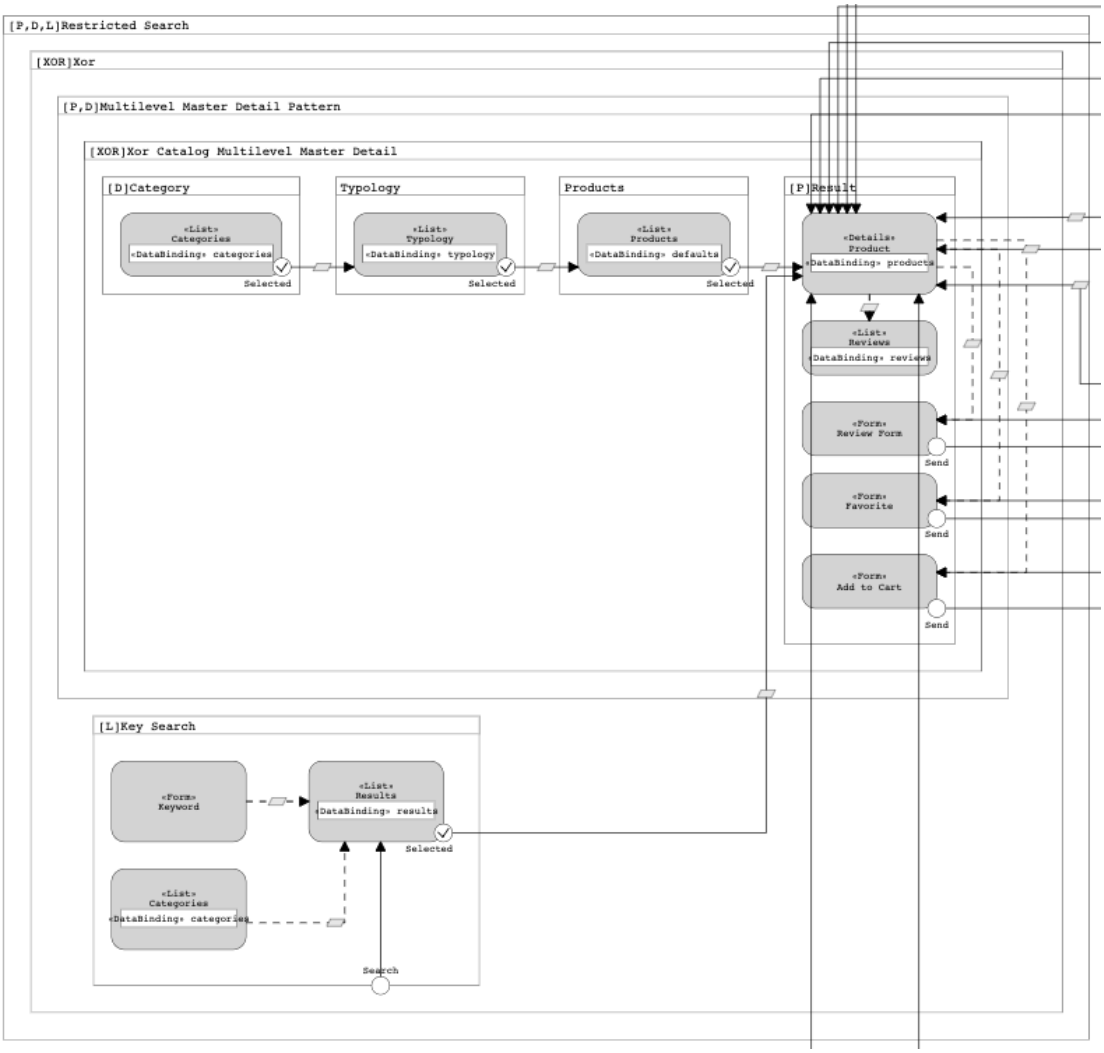


Figure B.5: E-commerce Model - Area 1, Restricted Search and Multilevel Master Detail design patterns

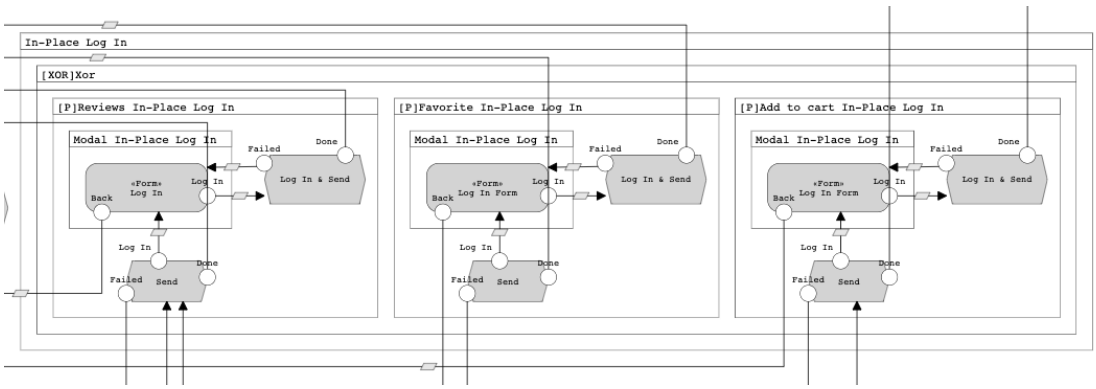


Figure B.6: E-commerce Model - Area 2, In-Place Log In design patterns

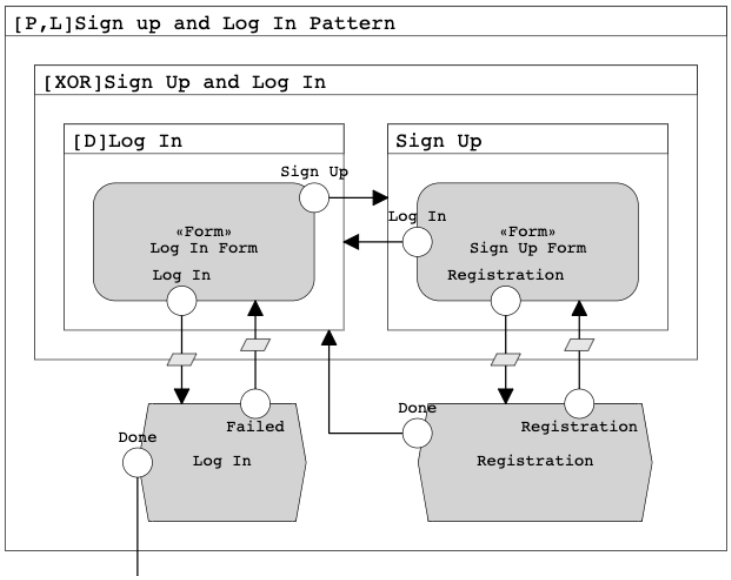


Figure B.7: E-commerce Model - Area 3, Sign Up and Log In design pattern

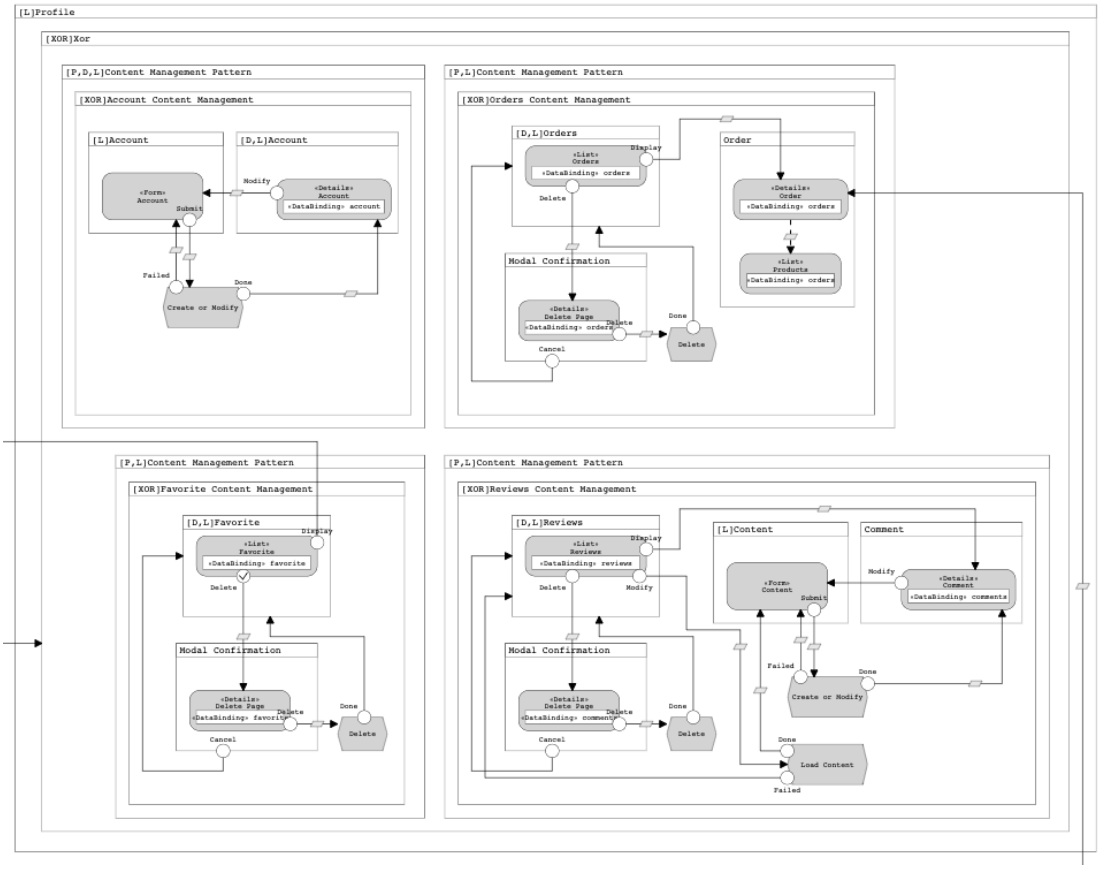


Figure B.8: E-commerce Model - Area 4 (first part), Content Management design patterns

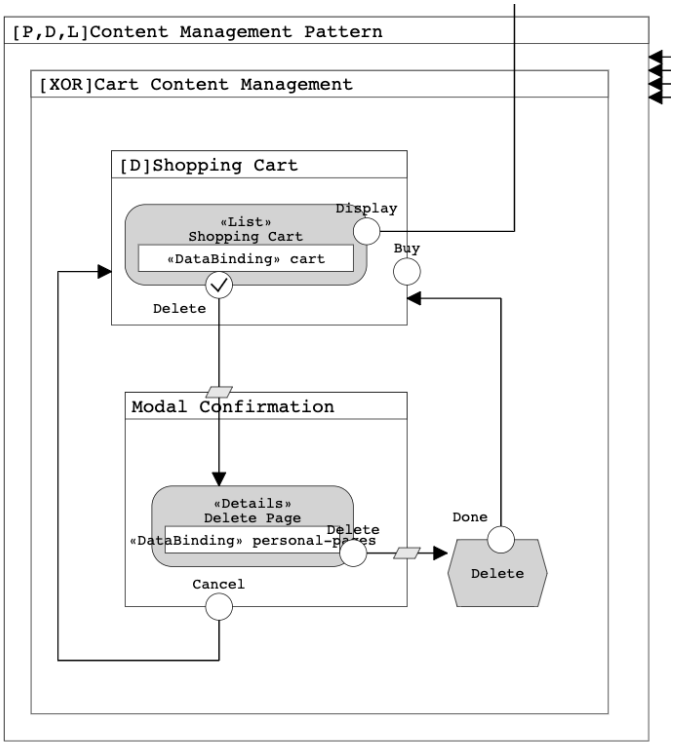


Figure B.9: E-commerce Model - Area 4 (second part), Content Management design pattern

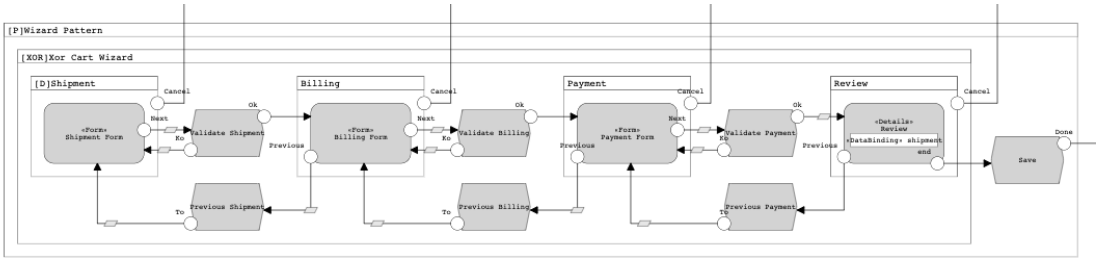


Figure B.10: E-commerce Model - Area 5, Wizard design pattern

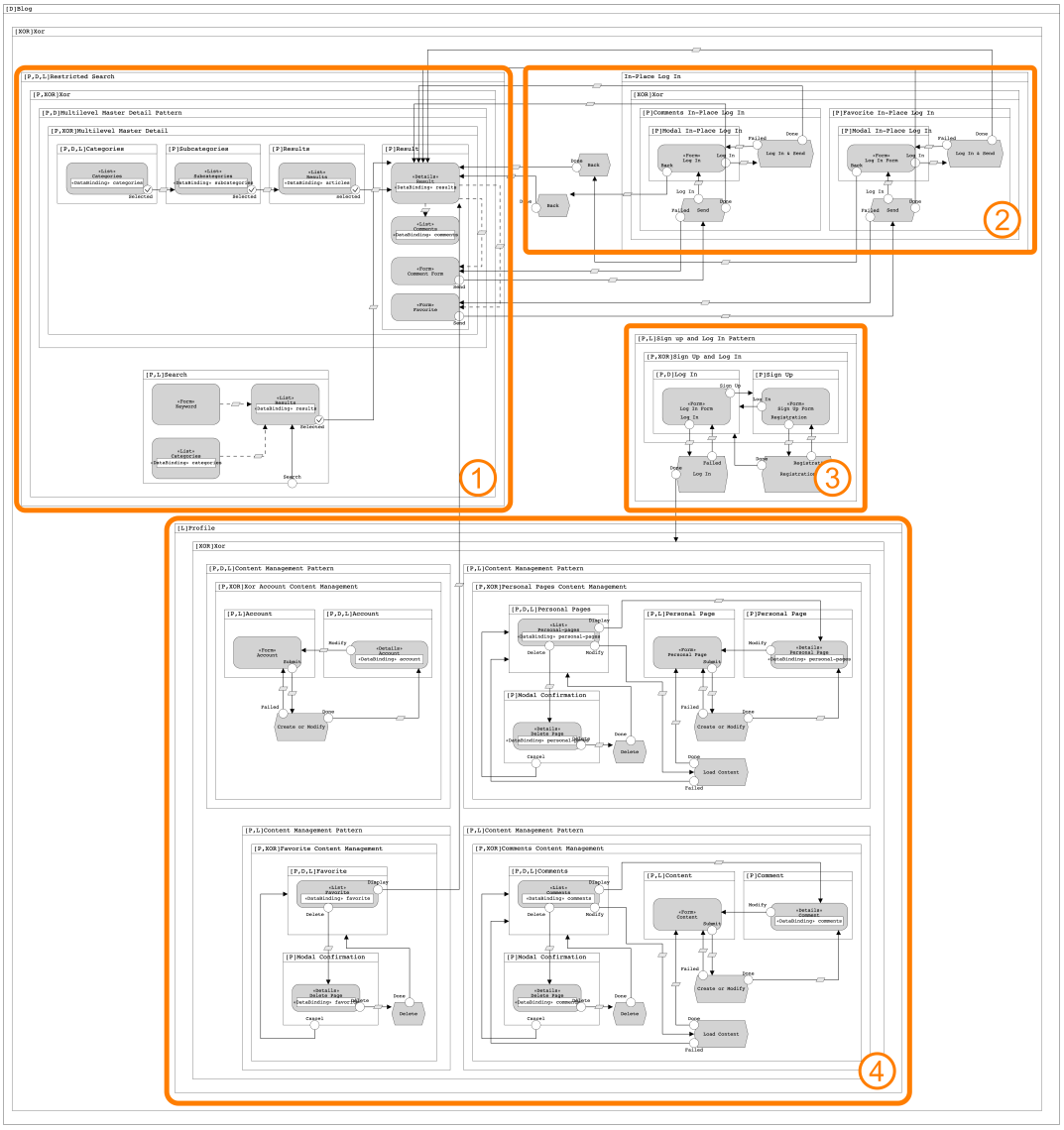


Figure B.11: Blog Application Model Example

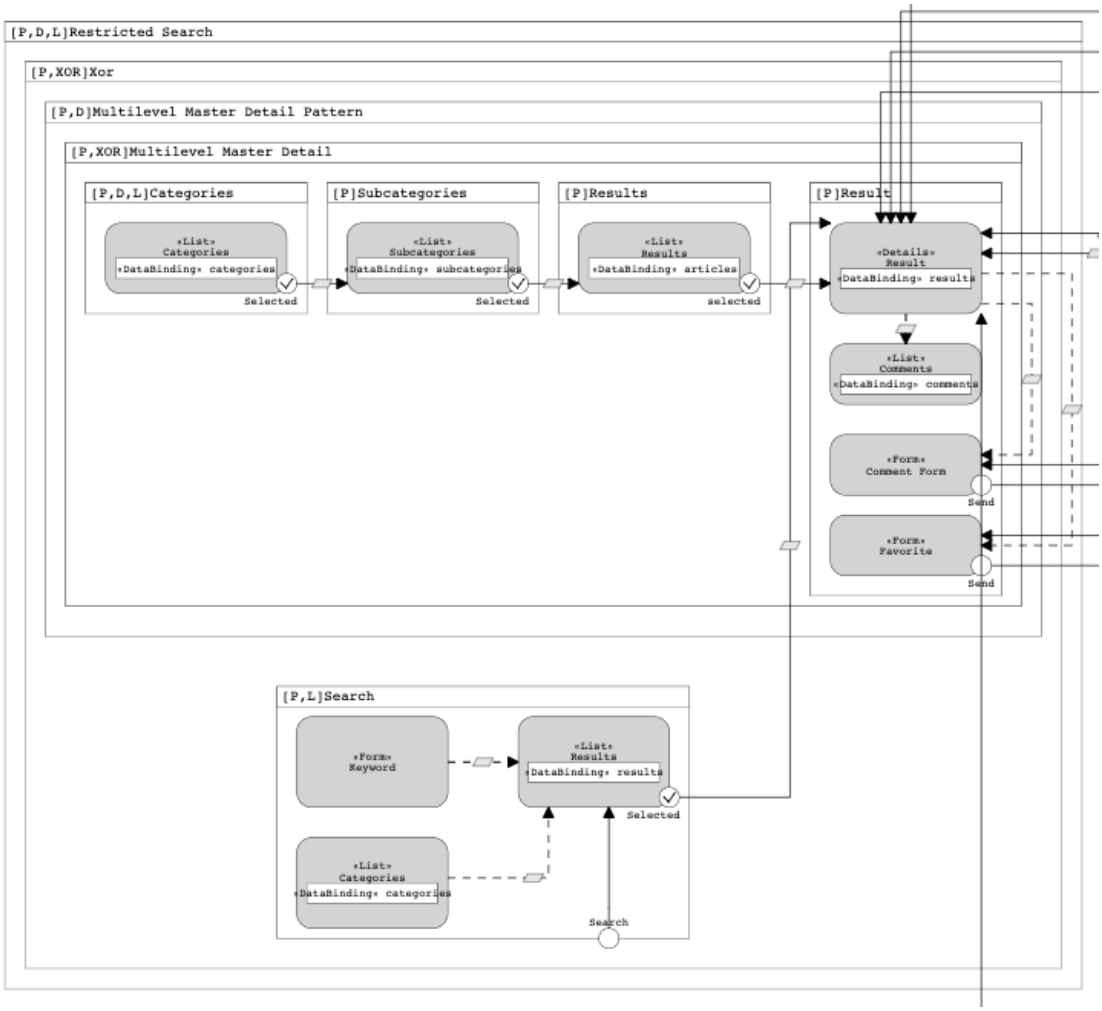


Figure B.12: Blog Model - Area 1, Restricted Search and Multilevel Master Detail design patterns

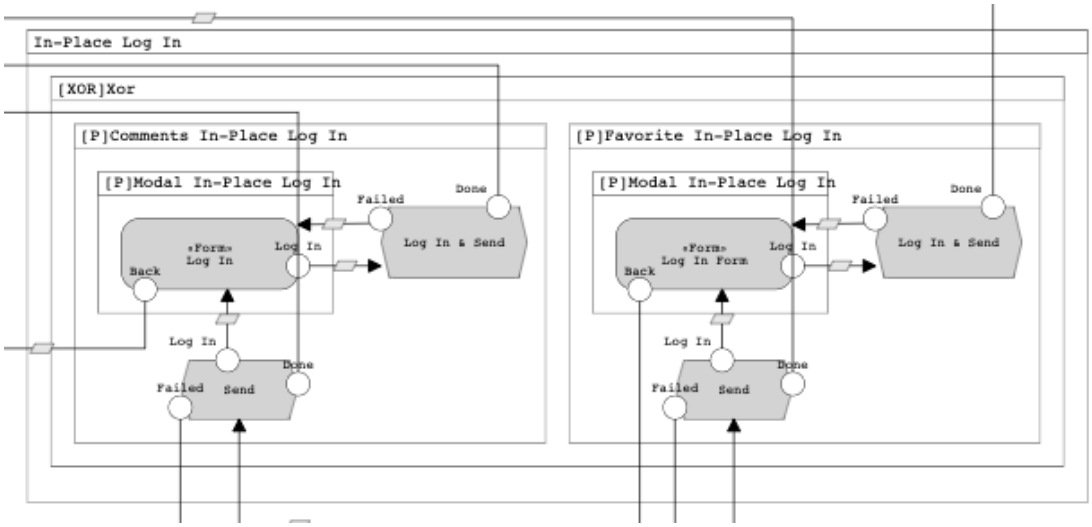


Figure B.13: Blog Model - Area 2, In-Place Log In design patterns

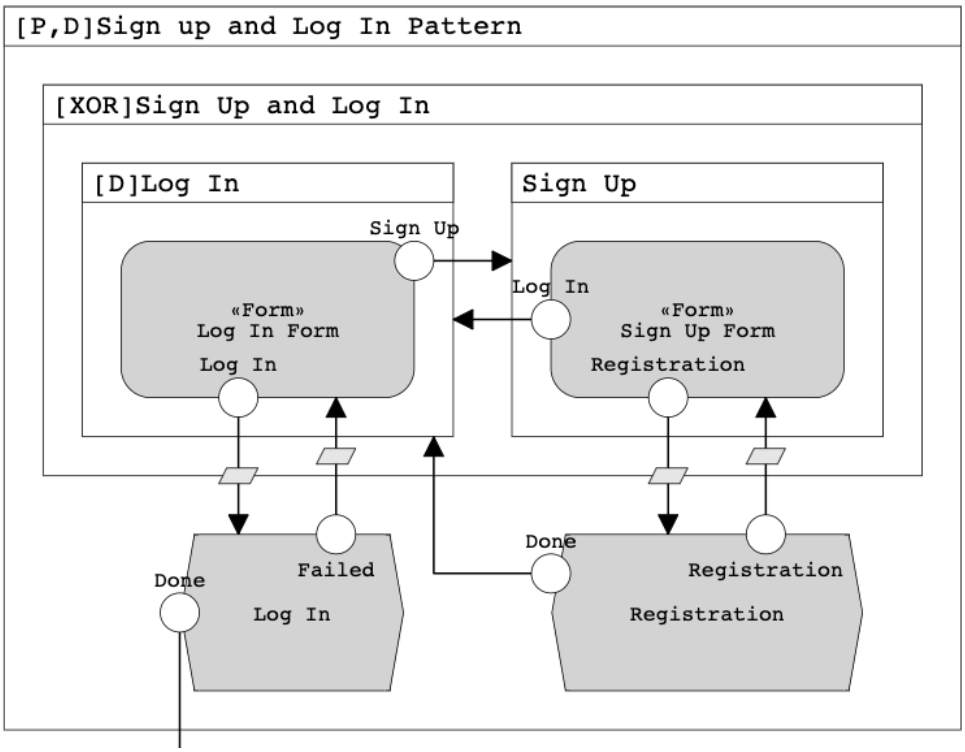


Figure B.14: Blog Model - Area 3, Sign Up and Log In design pattern

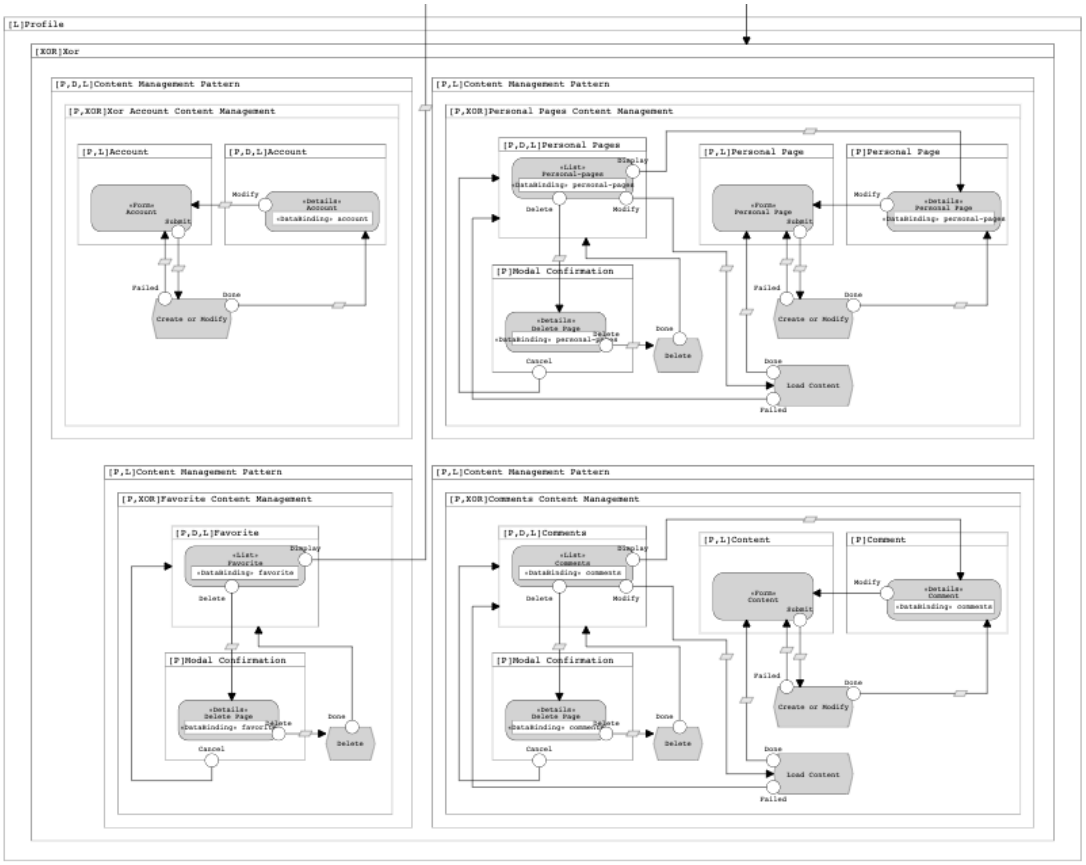


Figure B.15: Blog Model - Area 4, Content Management design patterns

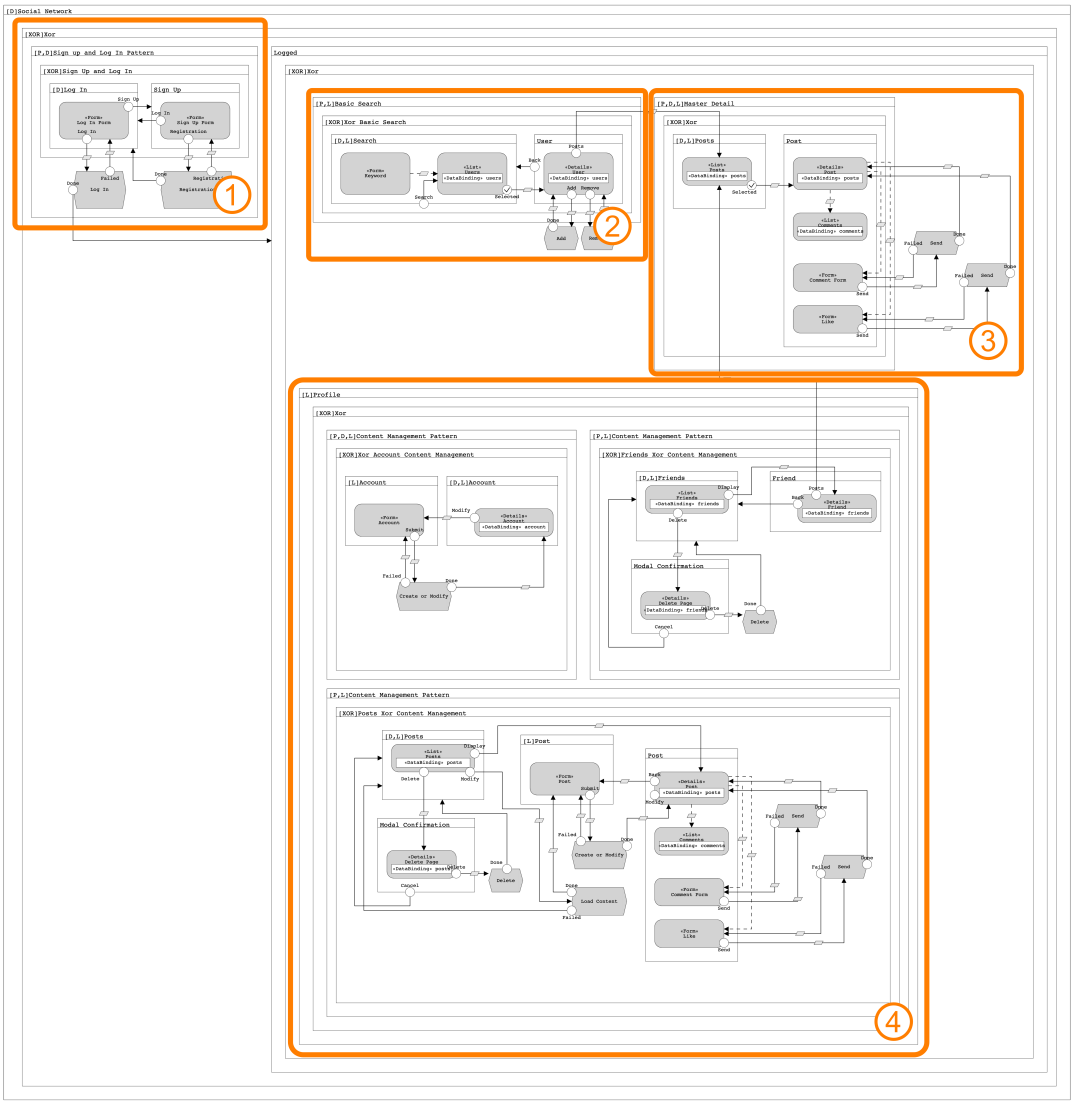


Figure B.16: Social Network Application Model Example

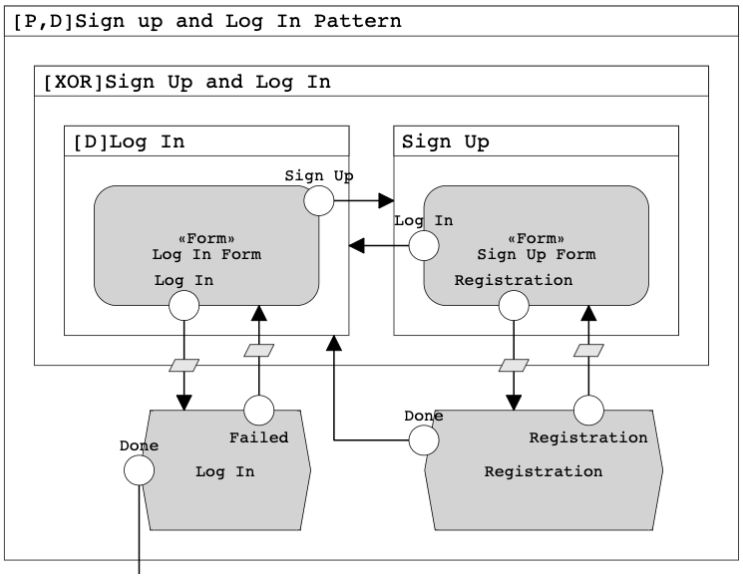


Figure B.17: Social Network Model - Area 1, Sign Up and Log In design pattern

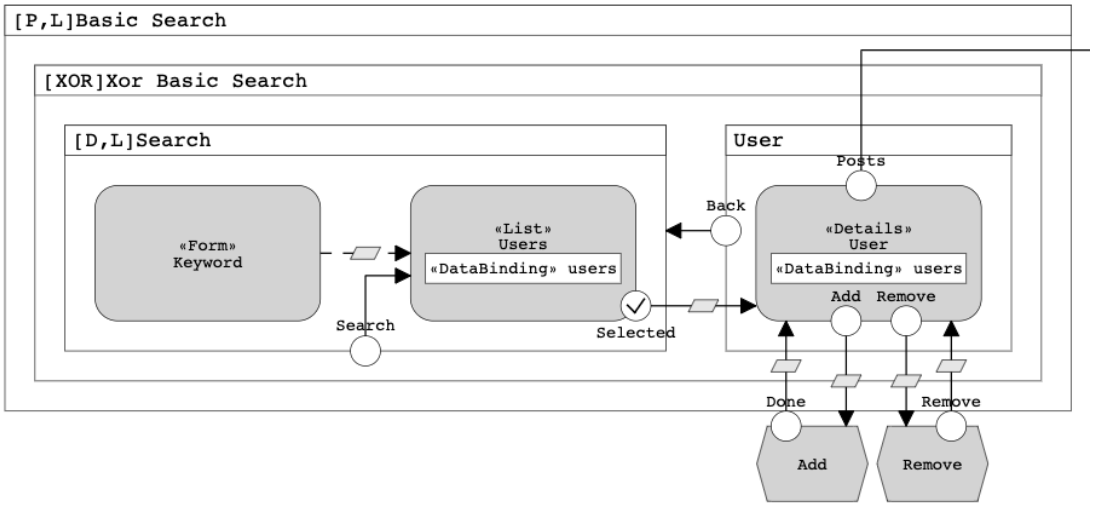


Figure B.18: Social Network Model - Area 2, Basic Search design patterns

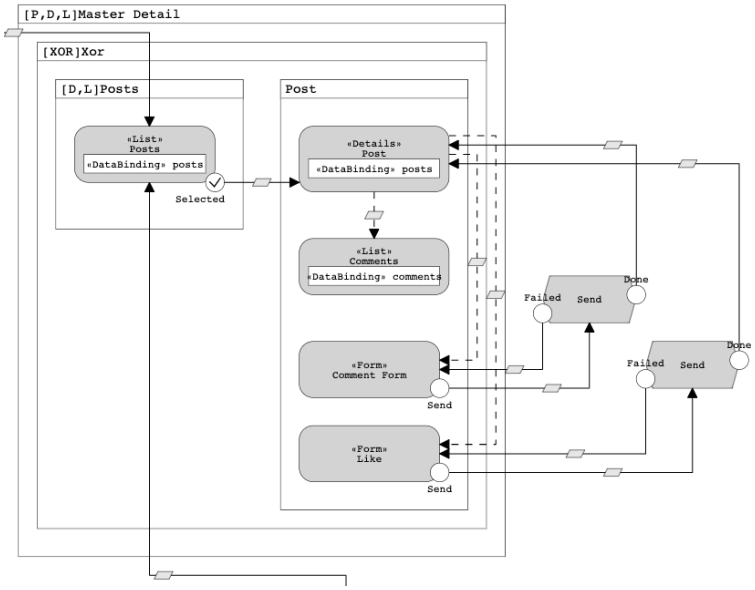


Figure B.19: Social Network Model - Area 3, Master Detail design pattern

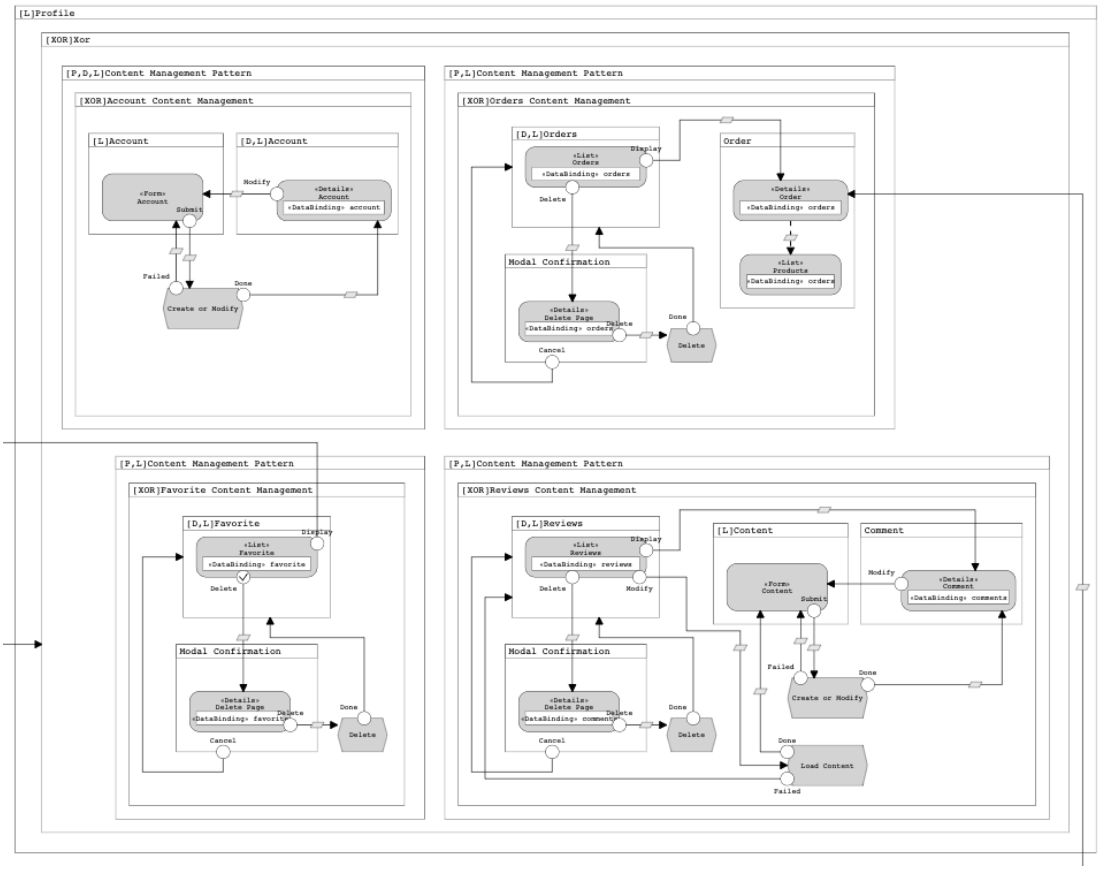


Figure B.20: Social Network Model - Area 4, Content Management design patterns

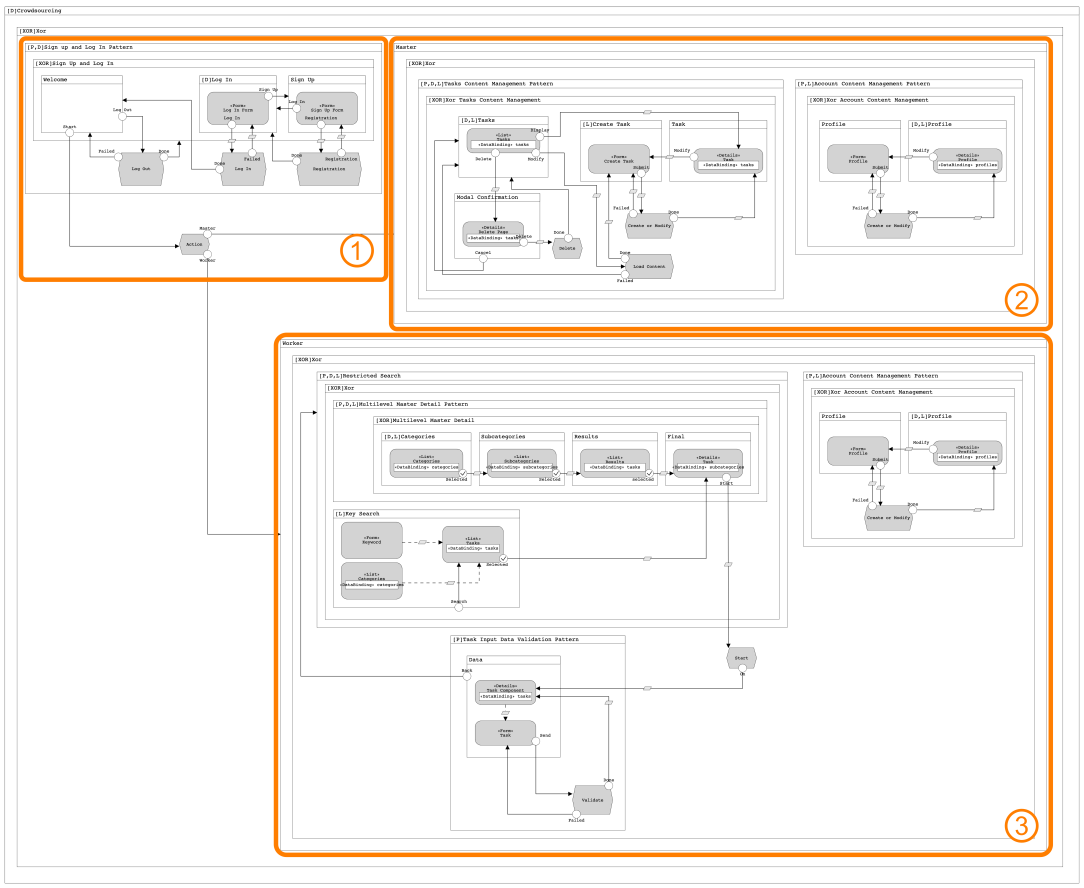


Figure B.21: Crowdsourcing Application Model Example

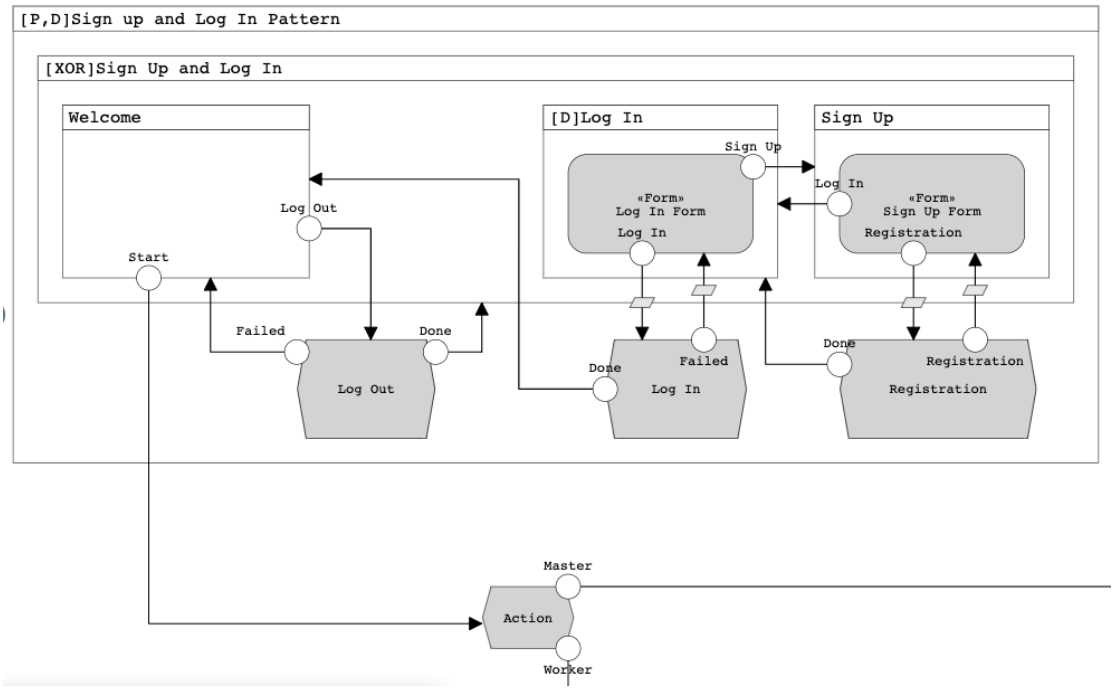


Figure B.22: Crowdsourcing Model - Area 1, Sign Up and Log In design pattern

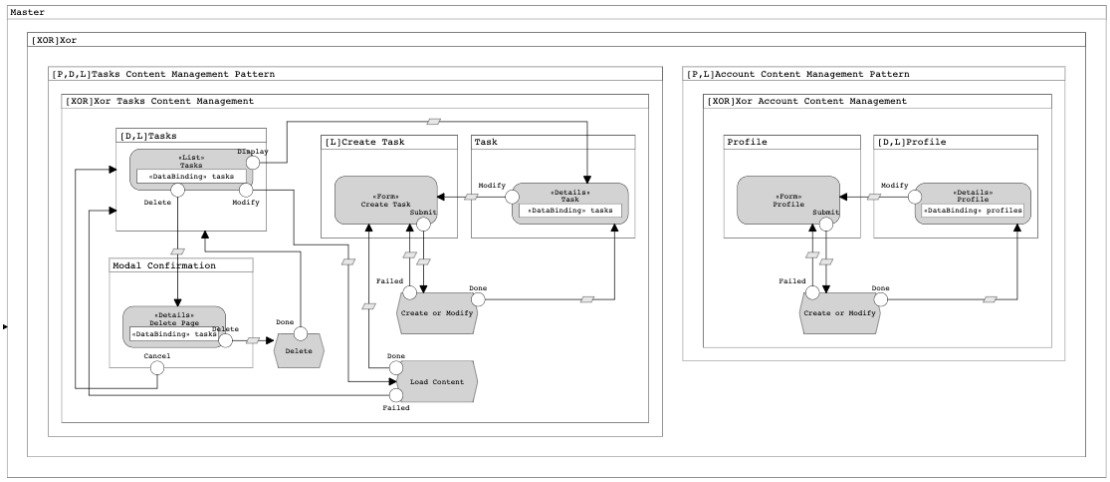


Figure B.23: Crowdsourcing Model - Area 2, Master profile

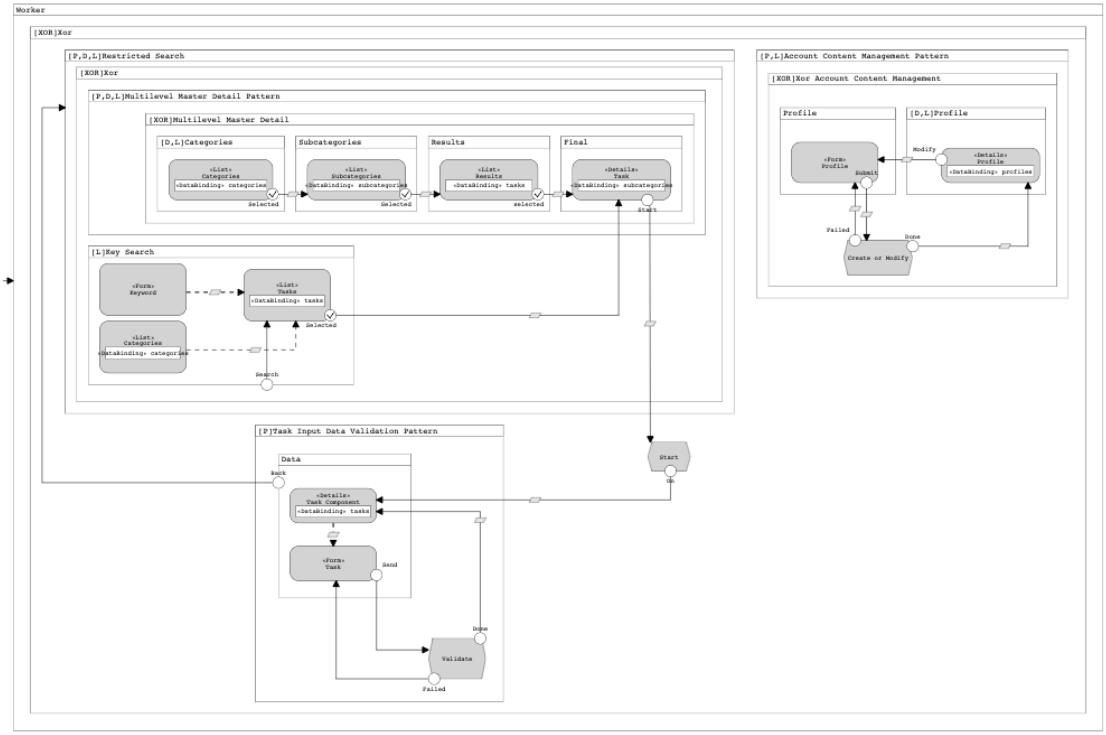


Figure B.24: Crowdsourcing Model - Area 3, Worker profile