The Journal of Systems and Software 117 (2016) 612–637

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Automated design of multi-layered web information systems

Fábio Paulo Basso[a,*], Raquel Mainardi Pillat[a], Toacy Cavalcante Oliveira[a], Fabricia Roos-Frantz[b], Rafael Z. Frantz[b]

[a] Systems Engineering and Computer Science Department, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil
[b] Department of Exact Sciences and Engineering, UNIJUÍ University, Ijuí, RS, Brazil

ARTICLE INFO

Article history:
Received 17 February 2015
Revised 10 March 2016
Accepted 25 April 2016
Available online 27 April 2016

Keywords:
Model-driven web engineering
Rapid application prototype
Domain-specific language
Prototyping
Automated design
Mockup
Experience report

ABSTRACT

In the development of web information systems, design tasks are commonly used in approaches for Model-Driven Web Engineering (MDWE) to represent models. To generate fully implemented prototypes, these models require a rich representation of the semantics for actions (e.g., database persistence operations). In the development of some use case scenarios for the multi-layered development of web information systems, these design tasks may consume weeks of work even for experienced designers. The literature pointed out that the impossibility for executing a software project with short iterations hampers the adoption of some approaches for design in some contexts, such as start-up companies. A possible solution to introduce design tasks in short iterations is the use of automated design techniques, which assist the production of models by means of transformation tasks and refinements. This paper details our methodology for MDWE, which is supported by automated design techniques strictly associated with use case patterns of type CRUD. The novelty relies on iterations that are possible for execution with short time-scales. This is a benefit from automated design techniques not observed in MDWE approaches based on manual design tasks. We also report on previous experiences and address open questions relevant for the theory and practice of MDWE.

1. Introduction

Model-Driven Engineering (MDE) (Kent, 2002) is a paradigm for model-based software development implemented by several techniques and used in several industrial contexts. In typical MDE-based processes, model transformations should receive a highly detailed model to generate working pieces of applications (Schmidt, 2006). To generate full source code, several parts of an application design are detailed in Domain-Specific Languages (DSLs) (Voelter, 2009) and/or decorated with annotations added to model elements represented with the Unified Modeling Language (UML) (Booch et al., 2005), a general-purpose modeling language commonly used. In any case, this makes the software construction dependent of design tasks.

In the development of web information systems, web front ends such as layout composed of Graphic User Interface (GUI) components (Vanderdonckt, 2005) and behavioral diagrams (Nunes and Schwabe, 2006) are usually represented. To allow the generation of full source code with an approach for Model-Driven Web Engineering (MDWE) (Rossi, 2013), these models are manually decorated with semantics for the actions of users, screen flows and business logic. It is possible to abstract implementation details using a design language, focusing on the specification of semantics in models that formalize the knowledge about software requirements (France and Bieman, 2001). Before the source code generation, these models can be further refined by designers, enabling clients to experiment an executable prototype in the end. This approach is known as *multi-view* (France and Bieman, 2001), and the model is created and enriched taking as input high-level abstractions of other models that map implementation details through model transformations.

The execution of a multi-view approach for MDWE may use design tasks that require months of work (Kulkarni et al., 2011; Zhang and Patel, 2011). Depending on the size of the software project and the adopted schedule in software process iterations, the effort invested in detailing models is seen as a reason to avoid the adoption of some of MDWE approaches (Whittle et al., 2013). Therefore, the ability to execute these tasks in short time-scales is a desirable feature in some contexts, such as in start-up companies (Rivero et al., 2014; Giardino et al., 2014).

* Corresponding author.
E-mail addresses: fabiopbasso@cos.ufrj.br, fabiopbasso@gmail.com (F.P. Basso), rmpillat@cos.ufrj.br (R.M. Pillat), toacy@cos.ufrj.br (T.C. Oliveira), frfrantz@unijui.edu.br (F. Roos-Frantz), rzfrantz@unijui.edu.br (R.Z. Frantz).

http://dx.doi.org/10.1016/j.jss.2016.04.060
0164-1212/

A possible solution to speed-up the modelling phase, thus helping in the execution of iterations in short time-scales, is the use of techniques for automated design (Linington, 2005; Batory et al., 2013). In this paper, we suggest the use of three different phases for constructing models for MDWE, namely: evolutionary, architectural, and functional. Models are based on the Model-View-Controller (MVC) architectural pattern (Evans, 2004). Although each prototyping phase is handled by some DSL and tools found in the literature, their integrated use is still a challenge in MDWE.

We present a methodology for MDWE named MockupToME Method, which includes tasks supported by (semi-)automated design techniques for some use case patterns (Molina et al., 2002). We extend previous contributions (Basso et al., 2014b), by detailing tasks and artefacts that include many DSLs, developed to support the design of many layers of MVC-based application models, and the tools associated with these tasks for automated design. We also summarized data collected from two software projects, the first considering mostly manual design tasks and the second considering the use of tasks based on automated design techniques.

A partially assisted design through Wizards was used in the first software project, with iterations planned for one month or more. In the second project, we used MockupToME Method with iterations planned and executed with one to two weeks. Both approaches are based on use case patterns of type CRUD (Souza et al., 2007), and use the same DSLs for representation of MVC-based application models, which are used in the end of a lifecycle for model transformations by the same source code generators. Differently, MockupToME Method includes DSLs and tools for designers to work in high-level of abstraction than in MVC-based application models.

The use of short iterations is a benefit observed in MockupToME Method, but not in our previous approach, i.e., in manual design of these models. The reasons why short time-scales are feasible in MockupToME Method has to do with the automated design techniques discussed in this paper. Thus, we also derived interesting research questions as a result from these two software projects.

The rest of the paper is organized as follows: Section 2, conceptualizes this work and Section 3 motivates this research; Section 4 exemplifies the representation of preliminary requirements, which are the input for the automated design approach introduced in Section 5; Section 6, describes the methodology, which is complemented in Section 7 with implementation details and in Section 8 with activities performed after the source code generation; Section 9, summarizes the two software projects, with lessons and insights for future research; Section 10, points out limitations; Section 11 presents the related work; and, finally, Section 12, reports on our main conclusions and possible future work.

2. Concepts

In the context of the development of web information systems, the following concepts are important for the understanding of this paper (Evans, 2004; Souza et al., 2007; Allier et al., 2015):

- **Model-View-Controller (MVC).** Is an architectural pattern (Parnas, 1994) frequently used in the construction of web information systems (Burke and Monson-Haefel, 2006). This pattern is important to modularize and structure the source code in three layers, thus facilitating the maintenance (Bosch, 2013) and avoiding the erosion of architectures as they evolves over time.
- **Conceptual model.** A class diagram composed of analysis classes, also named entities, which represents the *Model* layer of the MVC (Evans, 2004).
- **GUI Templates.** Facilitate the development of standardized structures for GUI (Han and Liu, 2010) allowing developers to focus on the logic layer, while layout details and actions are managed by a template engine. By means of templates, developers focus on the content that is placed inside a template structure.
- **CRUD.** A type of GUI template and an acronym for *create, read, update, and delete* (Souza et al., 2007) characterizing frequent set of use cases developed in information systems that allow to persist, retrieve and remove objects to/from a database. Different structures for CRUD can be used, and may include a specific GUI template.
- **Domain-Driven Design (DDD).** The *Model* layer is used to represent all the other application layers using a DDD approach (Evans, 2004). In MDWE, DDD drives the generation of a detailed MVC-based model, guiding the refinement of multiple layers associated with a particular use case scenario and a paper prototype.
- **Master/Detail.** A well-known concept among software developers, which allows the classification of use cases for *use case patterns* (Molina et al., 2002). These concepts of Master and Detail are well discussed in approaches for DDD (Evans, 2004) and the object oriented method (Molina et al., 2002).

The following concepts are important to contextualize our work:

- **Use case scenario.** Is one of possible flows from a use case (Sommerville, 2010) or user story (Landre et al., 2007). Use case scenarios are important both for design and for tests with clients (Sommerville, 2010), which evaluate models, prototypes and also the final version of an application piece with acceptance tests.
- **Paper prototype.** A hand drawing on a paper showing user interfaces with user interactions that represents use case scenarios (Sommerville, 2010). It is a software artefact represented in a high-level of abstraction than a mockup. A paper prototype is not a model, but a document usually associated with user stories specified in initial brainstorming meetings for the requirements elicitation. It is also called as pre-prototype (Davis and Venkatesh, 2004) and, sometimes, as throwaway prototype (Sommerville, 2010).
- **Mockup.** A model for a GUI, which is not possible to be fully implemented in functional prototypes (Blankenhorn, 2004; Rivero et al., 2014; Forward et al., 2012). In our understanding, mockups are abstractions in a high-level than the business logic needed in the development of web information systems, focusing on GUI components specification. Mockups may also be called sketches (Balsamic Mockups Company, 2015).
- **Round-trip engineering.** A set of activities aiming at synchronize generated source code with manually developed code (Mussbacher et al., 2014). It is performed automatically with the support of tools or, sometimes, manually, when it is required to update the model based on changes from source code.
- **Full source code generation.** Is the ability to generate 100% of what is designed, not 100% of all the application (Kelly and Tolvanen, 2008). Kelly and Tolvanen (2008) claim that full source code generation is a possible solution that mitigates the execution of changes in generated artefacts.

The Java platform is important for the implementation of web information systems and is divided in J2EE, J2SE, and J2ME editions. Burke and Monson-Haefel (2006) state that:

1. For the development of forms to desktop platforms, developers adopt J2SE and APIs such as AWT and Java Swing.

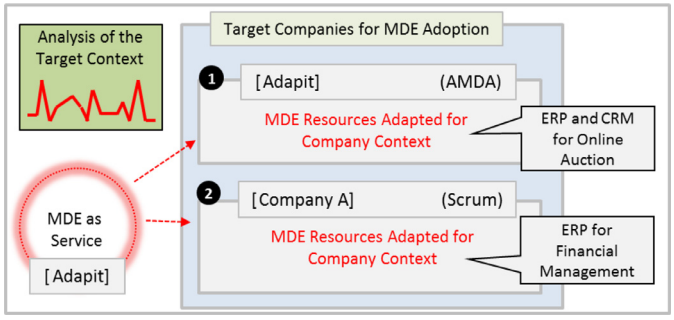

Fig. 1. Implemented scenario for MDE as Service considering contexts from two software projects, one executed by Adapit and the other by Company A.

2. For the development of forms to mobile platforms, developers adopt J2ME and APIs for MIDP such as Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC). The latter API focuses on devices that allow for rich GUI components and rich user interactions, whereas the former is quite limited. For this reason, CDC is supported by GUIs developed with the AWT, similarly as in J2SE edition.
3. For the development of forms on the web, developers adopt J2EE, which can include: (a) tools for the database management system such as PostgreSql; (b) web frameworks such as Spring Framework and frameworks for Object Relational Mapping (ORM) such as Hibernate; and, (c) APIs for the development of the *View* layer including JSP, JSTL, Dojotoolkit, jQuery, and so on.

Finally, some software projects may require all these editions in the development of multi-layered systems. This is the case of the systems that we have developed, which we discuss in the next sections.

3. Motivation and context

We have been in an effort to introduce model-based solutions in start-up contexts in an initiative for "MDE as a Service", as illustrated in Fig. 1. In this scenario, resources developed for MDE (e.g., model transformations, DSLs and tools) are applied in different contexts. We have implemented MDE as a Service by means of company *Adapit*, founded in 2007 and supported for three years by a business incubator, hosted in one of the biggest scientific and technological parks in Brazil. Through Adapit, we have introduced resources for MDWE in five software projects, three out of them developed by teams from Adapit and two out of them by teams from other start-ups.

The motivation for the advent of a new methodology and tool support came in 2007, from the internal application of our first approach for MDWE. It is a software project for the development of an web information system for online auction, hired on demand by another start-up, i.e., by an auction agency. This project needed the execution of iterations lasting one month due to distances between these start-ups. This time-scale implied in validations with clients carried out too late and, consequently, requiring a considerable rework in model and source code due to changes in requirements. Following the instructions from the software engineering discipline (Sommerville, 2010), we concluded that with shorter time-scales we could obtain feedback from clients in an early stage. However, due to a sum of factors such as the time invested in manual representation of models, issues in source code generation and bad practices for manual coding, hampered the execution of shorter time-scales.

As a solution to surpass these issues, between 2008 and 2010 we planned and developed an approach for automated design. It includes a tool named MockupToME and other DSLs in a methodology named MockupToME Method. It is the result of three years of industrial innovation, incepted exclusively for the application of MDWE in target software projects for web information system. Moreover, this approach is limited to assist the design of models for use case patterns such as CRUD, List, Filter, and Report.

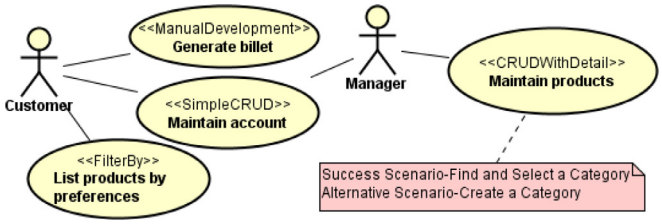

Fig. 2. E-commerce use cases designed manually conforms to the `CRUD UML Profile`. This is a use case view that illustrates a functionality considered for implementation in a unique iteration of the software development process.

We observed that MockupToME Method speed-up the specification of detailed MVC-based models within iterations planned in short iterations. In 2010 we implemented a feasibility study for MDE as a Service, by introducing our new approach to other start-up, referred to in this article as "Company A". Differently from Adapit, Company A adopts Scrum (Moe et al., 2010) as the reference model for the software development process. Likewise, we adapted our resources for the target context (Basso et al., 2013), analyzing issues associated with this specific reference model in combination with MDE (Basso et al., 2014d; 2015).

As illustrated in Fig. 1, our approach for MDWE has been used with different frameworks for management of software processes: Scrum and AMDA (Ambler, 2015). The MockupToME Method is agnostic to the framework adopted by the target software project and can be introduced in any model for software development process. We also represented our methodology with the BPMN (Pillat et al., 2015). However, it is also agnostic to the BPMN representation. Thus, the reader can consider it as flexible for inclusion of other tasks.

We present a contribution for the theory and practice of MDWE, discussing elements from methodology and tool support that configures our best approach for two start-up contexts. Likewise, considering mostly the worst-case scenario for design, we present some elements that we consider essential and optional for application of design techniques, tasks for validation with clients and coding issues. In the end, we also summarized a report of two systems, one developed with the automated design proposal and the other using mostly manual design (some wizards), and discuss on open questions associated with the MockupToME Method.

4. Running example

We illustrate our methodology considering the development of an e-commerce application, for the use case diagram shown in Fig. 2. Two actors, `Customer` and `Manager`, can perform the following use cases: (1) `Maintain Account`, which allows users to persist their personal data, preferences for categories of products and associate credit cards; (2) `Maintain products`, which allows users to persist data associated with products (e.g., a category); (3) `List products by preferences`, which allows customers to list products based on their preferences for categories; (4) `Generate billet`, which allows customers to pay for products using banking billet/slip. In the next sections, we demonstrate the automated design of the use case `Maintain products`.

The use case `Maintain products`, adopted for exemplification, also includes the following use case scenarios: `Success`

`Scenario, (1) - Find and Select a Category` - the category of a product is already available in the database, thus requiring the development of a complementary mockup to *Find* and to *Select* the category for inclusion in a product, or; `Alternative Scenarios, (1.1) - Create a Category` - the category was not found, it must be created, then the end-user re-execute the success scenario.

Although we focused on the exemplification of a form and filter for CRUD, it is also possible to automate the design of reports, lists, and other variations of user interactions with CRUD operations. Likewise, the unique use case that is not target for the presented techniques is the `Generate billet`, which requires manual development. The other use cases are possible to be automated similarly as the use case `Maintain products`. A reason for exemplification of `Maintain products` is its use case scenarios: besides the implementation of CRUD operations, the user can *Find* and *Select* the category for inclusion in a product.

Differently, the type of use case associated with `Generate billet` implies in the development of a scenario composed of the following implementations: (1) a mockup for list the products from a web shop kart; (2) a class to generate the PDF file. Due to limitations in our tool support to assist the design of `Generate billet`, the first implementation can be generated, but not the second one. Therefore, for didactic reasons, the reader should assume that `Generate billet` is manually developed, allowing us to exemplify the use of round-trip engineering.

5. Approach

Our MDWE approach is illustrated in Fig. 3 and includes the design of mockups with the MockupToME DSL[1]. A screenshot of MockupToME metamodel is shown in Fig. 4 (A). Models in conformity with such DSL are refined and transformed into other representations, thus following a multi-view design approach (France and Bieman, 2001). In the following sections we introduce our approach.

5.1. Tool support for the design

In a previous lifecycle adopted in 2007 for the development of the online auction system, the first representation adopted for the *View* layer was a representation in conformity with the GUI Profile (Blankenhorn, 2004), illustrated in a UML representation in Fig. 5. The literature recommends the usage of use case patterns and Master/Detail as a solution to facilitate the development of web information systems (Molina et al., 2002; Evans, 2004). To improve our previous practice based on manual design of MVC-based application models, we adopted this recommendation. Besides, for the sake of offering for designers a better conceptualization than the one available in GUI Profile (Blankenhorn, 2004), we developed the MockupToME DSL considering these recommendations (Molina et al., 2002; Evans, 2004). Thus, our DSL is introduced after the representation of the conceptual model shown in Fig. 6 and use cases, which are located at the top-part of Fig. 3 as the first representation associated with models in the lifecycle.

We found that the GUI Profile is limited to the representation of GUI components and does not require concepts for Master/Detail. This limitation was surpassed using MockupToME DSL, which is used as front end for the representation of GUIs together with relationships of Master/Detail. However, the GUI Profile is not discarded. Instead, we considered it as a generic DSL for GUI components that follows other representation specific of target platforms.

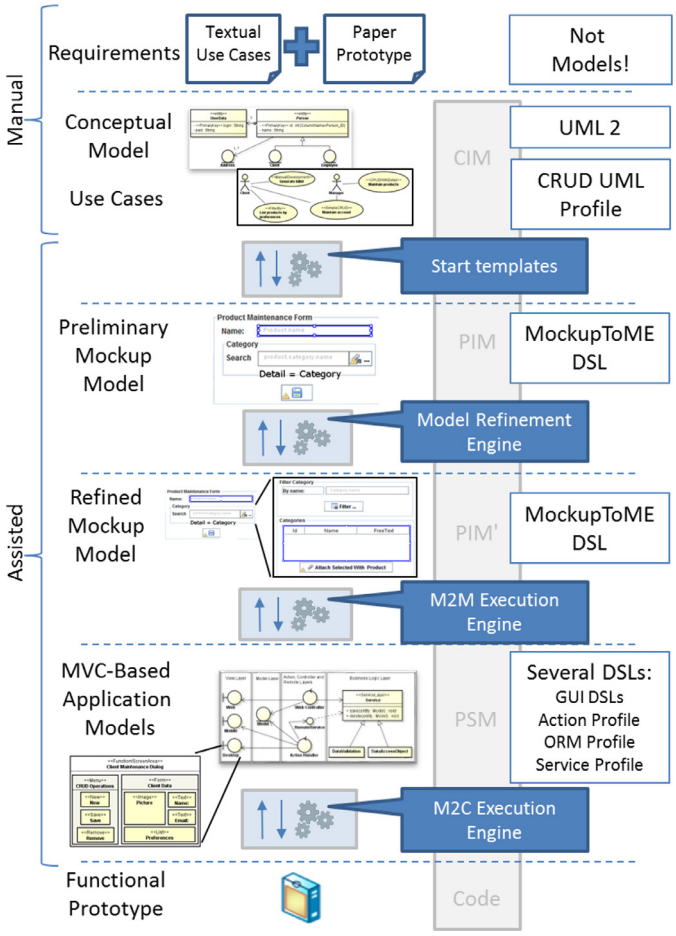

Fig. 3. Model transformation lifecycle adopted in the automated design approach.

In the lifecycle of our proposal, the GUI Profile is implemented through the metamodel illustrated in Fig. 4 (B) and belongs to a set of representations called `MVC-Based Application Models`. This DSL is included in the second level of representation for GUIs called `Concrete GUI metamodel`, which in fact is generic and built on top of other DSLs for platform dependent GUIs as follows: (a) Web DSL metamodel is illustrated in Fig. 4 (C) and allows the representation of details for components based on W3C/HTML 5[2]; (b) Mobile DSL metamodel is illustrated in Fig. 4 (D), which is based on the Java J2ME Components (Burke and Monson-Haefel, 2006) programmed in MIDP and CDC-AWT[3]; and (c) Desktop DSL metamodel is illustrated in Fig. 4 (E) allows the representation of details for components based on the Java J2SE/Swing components (Burke and Monson-Haefel, 2006).

HTML properties such as `css`, `class`, `background`, etc., can be represented in components from the Web DSL, which is not possible to be specified in components that conforms to the MockupToME DSL. The same is valid for Mobile and Desktop DSLs. To focus on the methodology, this paper does not provide details on such metamodels neither the conservatives UML extensions that belong to our UML Profiles.

To assist the representation of such models, our methodology includes tasks supported by (semi-) automated design techniques, which speed-up the specification of the detailed MVC-based models, allowing the use of iterations lasting one to two weeks. Thus, through an specification in conformity with MockupToME DSL, we

[1] MockupToME web page. Available at: <prisma.cos.ufrj.br/wct/projects/mockuptome_home.html>.

[2] <https://www.w3.org/TR/html5/ >

[3] <http://www.oracle.com/technetwork/java/index-jsp-138820.html >

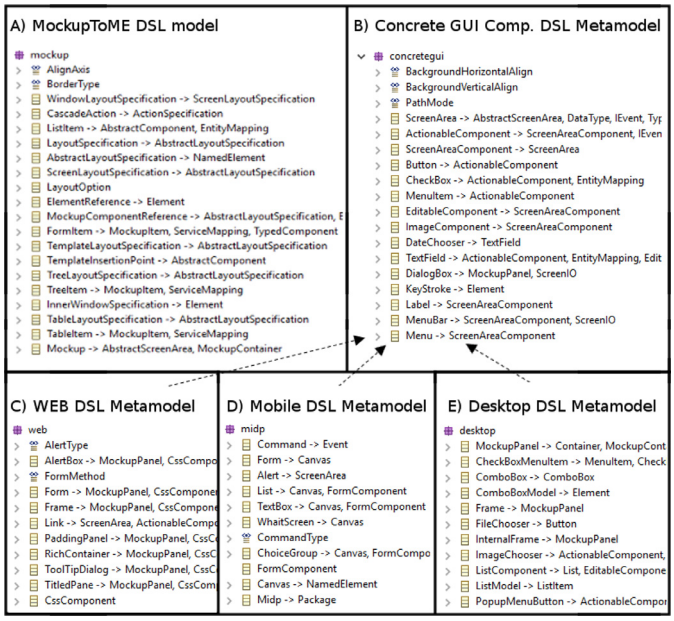

Fig. 4. DSLs for representation of the *View* layer in different abstraction levels.

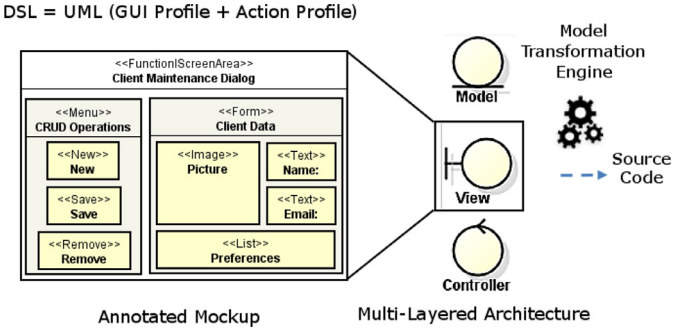

Fig. 5. Illustration of an annotated mockup, manually designed in the Astah UML modeling tool with stereotypes from the GUI Profile (Blankenhorn, 2004) and our extensions from `Action Profile`, as part of the view layer of a multi-layered architecture based on MVC.

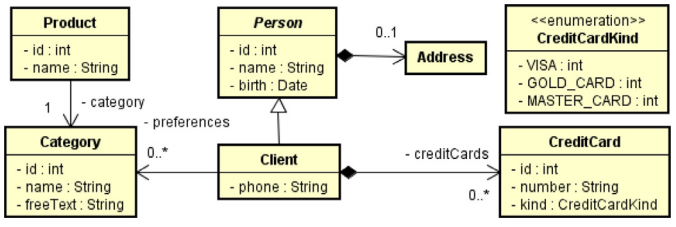

Fig. 6. Conceptual model designed manually with the Astah UML tool. This is a logical view of the *Model* layer, illustrated as part of the scenario associated with the use case view shown in Fig. 2.

included (semi-) automatic transformations from mockup models to other models in conformity with MVC-based architectures.

5.2. Lifecycle of model transformations

In the following we discuss some conceptual specificities used in Fig. 3:

Preliminary specifications. Are textual use cases/user stories and paper prototypes, the minimum input for our approach to automate the design in MDWE. These artefacts are not model specifications and serve as guide for the designer that works in a use case, such as those illustrated in Fig. 2. In preliminary software development phases, a requirement engineer draws user interfaces in a paper, based on use cases or user stories. The engineer is free to select techniques and tools to perform these tasks, such as use case augmentations (Ricca et al., 2010), inspections, and pre-prototypes (Davis and Venkatesh, 2004).

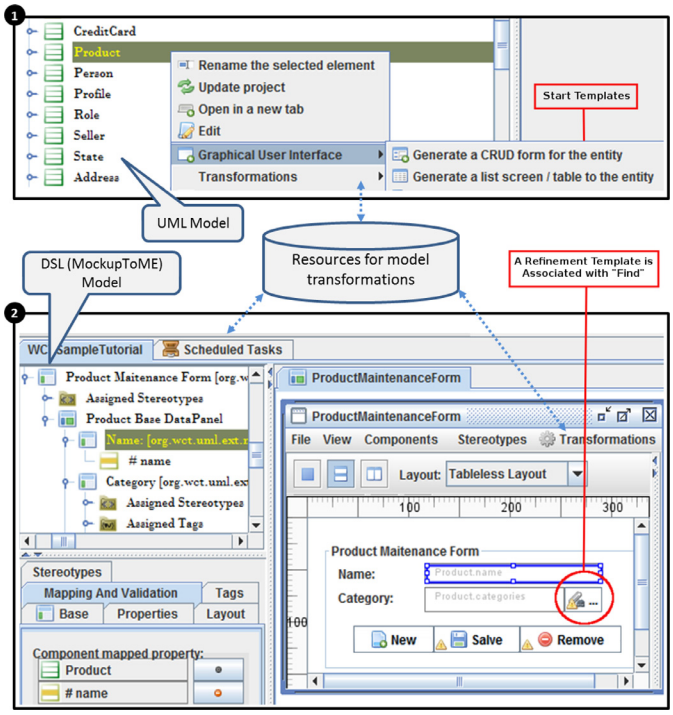

Fig. 7. Screen-shots of models and start templates.

Mockups are models. In the modelling of web information systems, a mockup is a GUI whose components are associated with operations for CRUD, data filter and reports (Ricca et al., 2010). Following the motivating example, Fig. 7 (2) shows the simplest structure for CRUD available in our prototyping tool[4] for the design of a mockup that implements the use case `Maintain products`. The design in a mockup is semantically connected with one or more layout structures, e.g., with a GUI template for CRUD, that are commonly used in the development of web information systems.

Annotated mockup. As illustrates Fig. 3, mockups are represented and refined in conformity with the MockupToME DSL. This language allows the representation of annotated mockups, as illustrated in the left-side of Fig. 7 (2) through tags and stereotypes. Through annotations, the proposed mockups own semantics for action, as in the proposals by Ricca et al. (2010) and Rivero et al. (2014). Mockup designs are annotated with semantics associated with standard actions, which are expanded in new mockups that implement the diverse scenarios of a use case. Thus, it is possible to infer the user interaction in these type of functionality, allowing to perform simulations in web browsers without the need to detail flows between GUIs.

Assisted design of models. In previous experiences we always looked for ways to speed-up the design of models, making them less dependent from specialists. For example, the set of artefacts found in our `MVC-Based Application Models` (see Fig. 3) are divided in some layers represented with specific EMF-based DSLs and UML Profiles. Fig. 5 illustrates some annotations based on the GUI Profile (Blankenhorn, 2004), applied manually for the view layer. However, several other layers and annotations from other

[4] A demo from MockupToME tool is available at: <https://www.youtube.com/watch?v=TrjuqLJMy8M >.

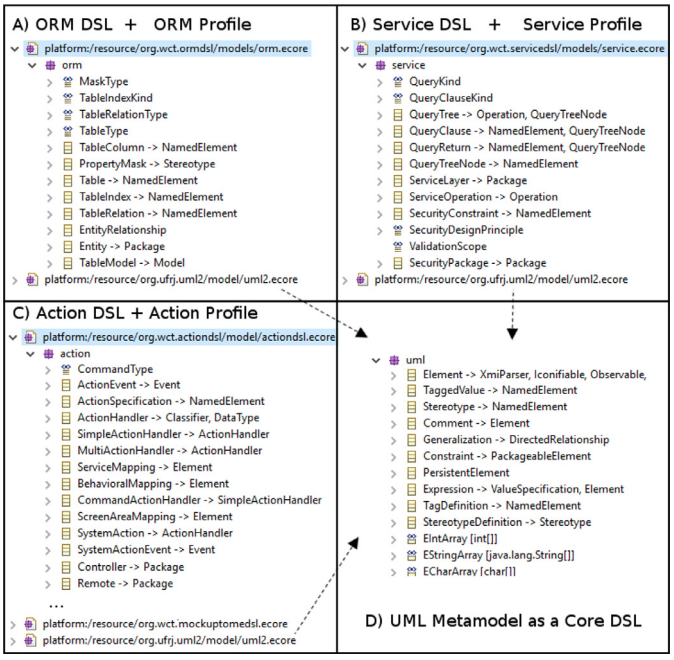

Fig. 8. Metamodels for the representation of `MVC-based Application Models`.

UML Profiles are necessary to represent the set of artefacts. This requires a considerable time for design that should not be ignored in software projects conducted with short iterations. Through the execution of the overall automated design lifecycle illustrated in Fig. 3, we can assist the representation of these model specifications, some generated automatically, refined by the designer in each phase of prototyping.

Start templates. The difference between template-based development and our approach is that templates are model transformations, thus not just as a source code facility. Likewise, we propose to use model transformations of type `start templates` to generate preliminary mockups. A start template is a classification of model-to-model transformations that allows the generation of a `Preliminary Mockup Model`, such as the one illustrated in Fig. 7 (2). Mockups embed the structure for one or more start templates. This is illustrated in Fig. 7 (1), which shows a start template executed against input entity classes designed in a class diagram shown in Fig. 6. This allows the automatic generation of the mockup shown in Fig. 7 (2). Therefore, a start template follows the same principles from GUI templates, but it is applied specifically to generate a model in conformity with the MockupToME DSL.

Refinement templates. A refinement template is similar, although strictly applicable to generate mockup structures for *Details* in representations for use case scenarios. For example, a *Detail* from the entity `Product` is `Category`. Considering the success use case scenario for `Maintain Products`, refinement templates are applicable to the entity `Category` in association with `Product`, allowing the generation of a `Refined Mockup Model` in the lifecycle. This is because a refinement template is associated with the *Find* pattern (see the button just above the `Remove` button in Fig. 7), which allows to generate automatically other mockups for *Search*. The generated mockup can also be refined in another specifications to *Create* a `Category` and so on.

MVC-based application models. An annotated mockup is an input for model-to-model transformations, that allows the generation of other specifications in conformity with DSLs, as illustrates Fig. 8, used to represent MVC layers, as illustrated in Fig. 9. Fig. 8 (A) illustrates the metamodel for the representation of Object Relational Mappings (ORM) (Burke and Monson-Haefel, 2006), the

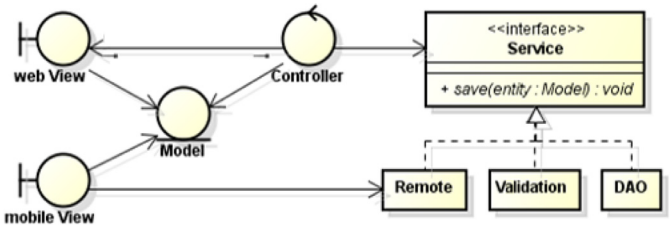

Fig. 9. Structures for multi-layered application based on MVC.

DSL for the representation of business logic and database queries is illustrated in Fig. 8(B) , and Fig. 8(C) illustrates the metamodel for the representation of actions for the *Controller* layer. These DSLs extend the UML metamodel shown in Fig. 8 (D), allowing the representation of annotations such as tags and stereotypes in conservative extensions. The Action DSL also extends the MockupToME DSL, allowing the connection between mockups and MVC-based Application Models.

Multi-layered MVC. Some companies promote the usage of more layers for better structuring the source code than those known in the MVC pattern (Allier et al., 2015). As illustrated in the bottom-part of Fig. 3, `MVC-based Application Models` are structured in multi-layers shown in Fig. 9 using some DSLs shown in Fig. 8. Likewise, apart from the `Model`, the `View`, and the `Controller` layers, our architectural models and the generated source code are divided in: (a) `Remote layer` - it is a UML Class whose operations are annotated with tags and stereotypes from the EDOC UML Profile (EDOC, 2014), used to integrate business logic in a web server with client applications such as mobile and desktop; (b) `Validation layer` - it is a UML Class whose operations contain semantics for server-side logic to validate entities and properties, i.e., persistence constraints and regular expressions represented with metaclasses such as `PropertyMask` and `MaskType` shown in Fig. 8 (A); (c) the `Data Access Object (DAO) layer` - it is a UML Class whose operations are conformity with **ServiceOperation** shown in Fig. 8 (B), which allows the representation of semantics to apply database queries from CRUD-related actions.

Functional prototype. A functional prototype is a fully implemented prototype that can be tested in iteration cycles of acceptance with clients. In a multi-view design approach, a functional prototype is obtained through the representation of models in a Platform-Specific Model (PSM) view (France and Bieman, 2001). Thus, our functional prototypes are generated after the architectural prototyping phase, after mapping mockup designs for an MVC-based model. A functional prototype is result from model-to-model transformations, manual model refinements and generation of source code through model-to-code transformations.

WCTSample. This is a web framework that implements a multi-layered architecture. The framework has 18 basic entity classes to support access control, customizable CRUDs and filters, functionalities to handle files, and images that are common features in many web information systems. This framework was used in the development of the two systems reported in Section 9.

5.3. Final remarks

Although we focused and exemplified the design for a complex use case scenario associated with `Maintain products`, it is also possible to generate CRUDs for simpler scenarios (Basso et al., 2015). For example, to persist a `Category`, the resultant mockup could be as simple as the one illustrated in Fig. 7 without any information of *Detail*. Thus, the designer uses a start template and ignores the refinement templates. Moreover, for the design of simple mockups, some tasks included in our methodology

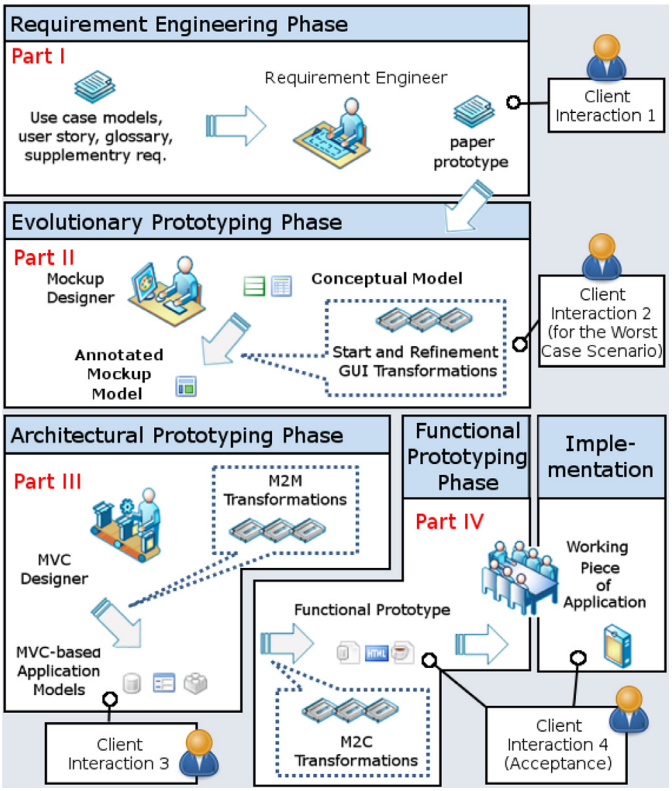

Fig. 10. Overview of steps proposed for rapid application prototyping in four interactions with client in the MockupToME Method.

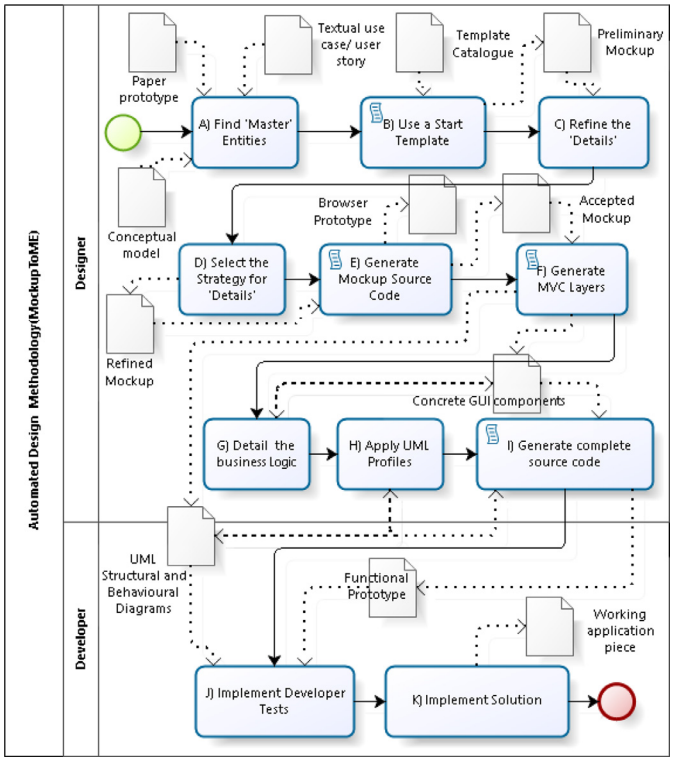

Fig. 11. MockupToME Method with tasks assisted by tool, allowing the generation of models in different abstraction levels.

can be ignored or partially used. Besides, it is always possible to represent components in mockups manually, detailing use case scenarios with annotations that are not supported in MockupToME tool. Thus, our contribution for automated design of mockups is complementary to manual design techniques introduced by Brambilla and Fraternali (2014),Rivero et al. (2014), and Ricca et al. (2010).

6. MockupToME method

Models are represented in different abstraction levels following a multi-view lifecycle in the MDWE scenario shown in Fig. 3. This approach is discussed from the perspective of stakeholders interacting with the MockupToME Method, as illustrated in Fig. 10.

Our methodology allows to work with four abstraction levels of artefacts associated with user interfaces: paper prototype, mockup model, concrete GUI models (platform specific), and functional prototype. Because we use more than one DSL in our approach, its systematization requires the following four different phases for prototyping:

1. Paper prototyping, which is executed in a requirement engineering discipline and represents the first view from the client about a functionality to be developed.
2. Evolutionary prototyping (Sommerville, 2010), which considers the worst-case scenario about the uncertainty of requirements as those found in start-up contexts (Giardino et al., 2014). This phase targets the exploratory development (Schwaber, 1995) of mockups with different options for clients to evaluate and decide which ones have to be used in his/her applications.
3. Architectural prototyping (Allier et al., 2015), which explores models that represent the MVC layers besides the View such as business logic, object relational mapping, and property validators.
4. Functional prototyping (Sommerville, 2010), which is the implementation of the source code for a functional prototype in which clients can perform acceptance tests.

Not all tasks presented in this methodology are mandatory. Thus, the software engineer must decide in each task about optional elements, such as the representation of alternative mockups for the implementation of a given use case scenario. The design of mockups for some use case scenarios associated with CRUD can be complex, involving a set of specifications for GUIs, actions and entities that should be represented in synchrony. For example, the use case `Maintain products` illustrated in Fig. 2 includes at least two scenarios that should be implemented: `Find and Select a Category`, or `Create a Category`. These semantics for actions are commonly found in use case scenarios for the development of CRUDs and present a standard workflow. Likewise, we found interesting to assist the design and refinement of these scenarios through automated design techniques.

Fig. 11 presents our methodology in BPMN. It is used in every iteration by a *designer* and *developer* to perform many cycles of validation, allowing iterative and incremental steps towards the development of working pieces of application. The *designer*, which is a specialist in mockup and MVC, refines a generated mockup model choosing mutually exclusive mockup structures from Tasks *A* to *D*.

Tasks *A* and *B* are fully executed independently of the complexity of the use case. Considering the worst-case scenario for use cases, Tasks *C* and *D* are fully executed. In these tasks, different structures of GUI components can support alternative implementation strategies, for example, different components, layouts and GUI templates for the same use case scenario. Finally, mockups are refined to support new suggestions.

A first executable prototype is obtained in Task E: `Generate Mockup Source Code`. The generated prototype is evaluated by clients to ensure that GUI flows and forms are in conformance

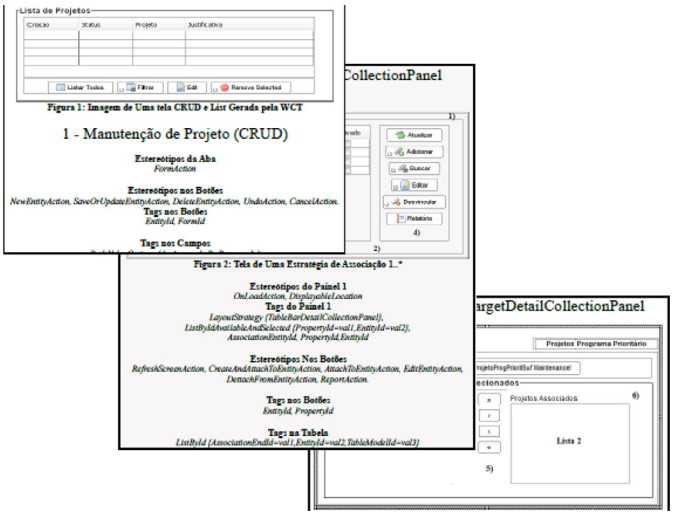

Fig. 12. Template catalogue, used as a guidance for execution of an assisted design approach through the MockupToME tool.

with the expected behavior to a given functionality specified in a textual use case scenario. Task F starts the `Architectural Prototyping Phase`, including the generation of models annotated with tags and stereotypes for business logic. An accepted mockup model is transformed into a concrete GUI model that is composed of DSL components supported by specific web technologies and APIs, e.g., an image chooser component. Concrete GUI components are refined in multiple views for each target platform.

A functional prototype with is generated from Tasks F to I shown in Fig. 11. The source code is changed to adjust details, and usability tests are executed by clients. Differently from the first executable prototype that supports only the simulation of flow screens, the functional prototype is fully implemented in MVC layers.

Tasks J and K, performed by the `developer`, are discussed in Section 7 and 8.

6.1. Part I: requirement engineering phase

Use cases and paper prototypes are elicited in late phase of the software development process and are used as input to decide whether and how the MockupToME tool should be used to automate the design of mockups. Our methodology starts, in fact, with a planning performed by a *requirement engineer* after the first *client* feedback from the designed paper prototype and use case.

In case of acceptance from these initial requirements, the requirement engineer will decide if the inputs are target for our automated design approach. This is possible due to a catalogue of templates that give instructions for design of some structures for CRUDs, List, Filters, and Reports named `Template catalogue`. This catalogue is illustrated in Fig. 12 and presents screen-shots of GUI structures for each classification of use case patterns together with annotations for Master/Detail. A template catalogue is used by designers to decide which start and refinement templates must be used for the assisted design of a mockup model. The design is performed considering a paper prototype, making a semantic association among these three artefacts: template catalogue, paper prototype, and mockup model. The mockup is the unique model from these artefacts, thus this association is not physically established among them.

The generation of a preliminary mockup occurs by means of `Start templates`. In this task the engineer semantically links use cases with start templates. Differently from the previous case, this link is physically established through a property of metaclass `Mockup`, available in the MockupToME metamodel illustrated in Fig. 4. This allows the connection of representations in conformity with MockuptoME DSL and UML, as shown in Fig. 8 (D).

6.2. Part II: evolutionary prototyping phase

In this section, we include a systematization of the usage of MockupToME tool in our methodology, by describing the interactions of the client/product owner with the designed mockups and its construction.

Fig. 11 illustrates the methodology that automates the tasks between requirement analysis and source code generation, and this section systematizes such tasks. To perform these tasks, end-users are assisted by tutorials and supported by tools discussed along the next sections, in which each task is detailed with: (a) artefacts represented as input and output; (b) a description of the associated model-based tool for design, refinement or transformation; (c) client interactions with the artefacts; and, (d) exemplifications.

6.2.1. Task A: *find* master *entities*

Input: Textual use case, Use case diagram, Class diagram, Paper prototype, Template catalogue.

Output: Master entities are included in textual use cases and related with a use case diagram using a tag. This is required to keep traces between artefacts.

Description: After a textual use case is elaborated, the *designer* analyses the domain classes looking for those that are characterized as master entities by the domain-driven design (Evans, 2004) and the object oriented method (Molina et al., 2002). Based on the paper prototype, the designer selects the *Master* entity from a class diagram shown in Fig. 6, for each use case to be developed from the use case diagram shown in Fig. 2.

Exemplification: After identifying the *Master* entity, the designer accesses inputs from Task A to identify which of the templates from the `Template Catalogue` is more adequate to start a design of GUI form. In order to automatically generate a mockup of type *form*, MockupToME takes as input a domain class diagram and, optionally, use cases. Forms are automatically generated through Start templates, selected in conformity with the use case scenario selected for development. For example, use cases shown in Fig. 2 stereotyped with ⟨⟨FilterBy⟩⟩, ⟨⟨SimpleCRUD⟩⟩, and ⟨⟨CRUDWithDetail⟩⟩ are target for start templates, described in the artefact `Template Catalogue`. Use cases are not mandatory for the generation of a mockup because they are used only to instruct and document, differently from the *Master* entity. Thus, to design the solution for the use case `Maintain product`, the designer will use the start template `Generate CRUD form`, activated on the entity `Product`, as shown in Fig. 6. The execution of this task is illustrated in Fig. 7 (1).

6.2.2. Task B: use a start template

Input: Textual use case, Use case diagram, Class diagram, Paper prototype, Template catalogue.

Output: Preliminary mockup.

Description: As in some web frameworks, many templates are available as a facility for codification of CRUDs: some are used to generate a mockup based on forms, other ones for list and filters, others for reports, between other structures. Instead of code facility, a `start template` is facility for the generation of preliminary mockup models. The artefact `Template catalogue` illustrates for the designer possible structures for generation of a `Preliminary mockup`. Likewise, this task aims at deciding which `start template` is directed for the generation of a mockup that must be developed in each iteration of the software development process. Thus, the designer choose the `start`

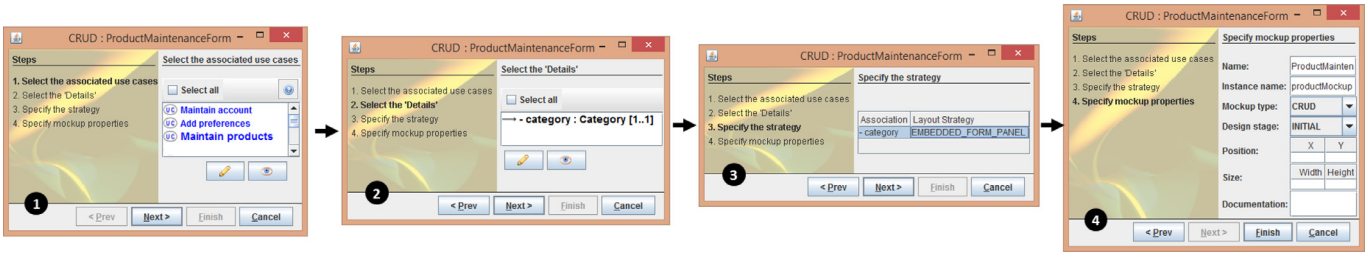

Fig. 13. Wizard executed by start templates.

`template` that best meets the structures drawn in the `Paper prototype`.

Exemplification: The activation of start templates will display the wizard illustrated in Fig. 13: **(1)** the execution of the first step named `Select the associated use cases` establishes automatically a link between the mockup with the selected use case. Note that it was selected the use case `Maintain products`; **(2)** The second step is to select the associations of the master class `Product` that will be included in the preliminary version of the mockup, i.e., the *Details* that will be included into the preliminary mockup. It was selected the association with `Category`; **(3)** The third step is to configure for each selected association a *Layout Strategy*. A layout strategy is a template for *Details*, and it is independent from start templates and is used to provide a particular layout that will handle the selected association, thus implementing the relationship between Master and Detail. It is important to note that, as long as the paper prototype represents exactly what the client needs, this step is effective because the selection of a layout strategy will generate the mockup with the structure as represented in the paper prototype. However, the effectiveness of step 3 is not always true, which imply in some cases in which the client will request another layout strategy. MockupToME is ready for this situation, allowing changes after the execution of start templates through refinement templates (see Task C); **(4)** The last step is to specify some properties of the mockup that will be generated after the mockup designer activate the button `Finish`.

Considerations: For the generation of a mockup without details, the designer should not select associations in the step 2. The non selection of at least one association will make the step 3 unnecessary. The result is a `Preliminary Mockup Model` with or without details.

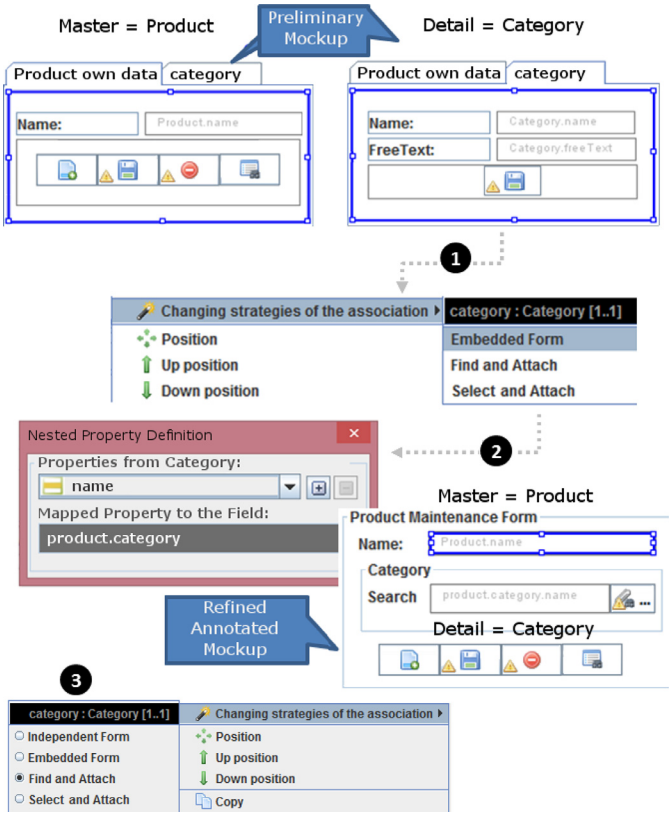

Fig. 14. Steps to refine annotated mockups: changing refinement strategies to handle associations.

6.2.3. Task C: *refine the* details

Input: Preliminary mockup, Master and Details, Textual use case, Paper prototype.

Output: Refined mockup.

Description: The refinement of the preliminary mockup is exemplified in Fig. 14. The goal is to reach the representation of a `Paper prototype` in a mockup specification through refinements. This can happen if the designer selects wrongly the strategy for a *Detail* in the screen (3) from the wizard shown in Fig. 13 or due to changes requested by the client/product owner. Likewise, after the generation of a preliminary mockup, the designer can change, if needed, the structure used to persist the *Detail = Category* inside a CRUD for the *Master* entity (`Product`), using other alternative structures for Master/Detail. In case these changes of structures are not necessary, then the designer apply adjustments in the mockup specification and follows to Task E. In the worst-case scenarios, where the designer is not 100% sure about the acceptance of preliminary requirements represented in textual use case and paper prototypes, this task may introduce alternatives for implementation of a use case scenario. Thus, the goal in this task is, in the worst-case of a software project that presents some uncertainty on the preliminary requirements, the generation of alternative structures to support the same master/detail relationship.

Exemplification: In the current format, a product and its category are persisted in different transactions, each one having its own `Form` and `Save` button, as illustrated in the top-part of Fig. 14. This structure should be changed by the one illustrated in the bottom-part. Our tool facilitate the application of this refinement. The transformation of one structure into another is an easy task since model transformations are available in pop-up menus executed over each of the elements of designed mockup shown in Fig. 14 (1). Using the drawing area one can undo transformations to decide what strategy best fits to express a specific part of functionality. Assume that the mockup designer selects the option `Find and Attach` in the second step and that panel titled `Category` with a component to *Find* something is automatically generated and configured. In this example, a refinement generated two mutually exclusive *Layout Strategies* i.e., Fig. 14 (1) and (2).

Strategies to refine *details* in associations of type (0..1): The use of different strategies to handle the same master/detail relationship are shown in Fig. 15. Given that compositions be-

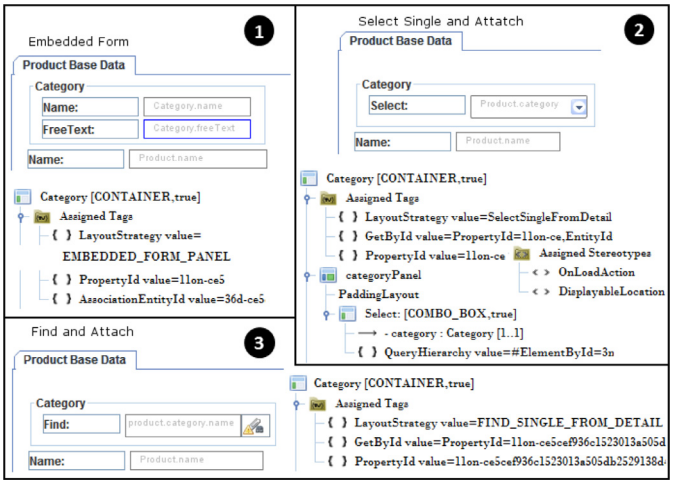

Fig. 15. Layout strategies for 0..1 relationships.

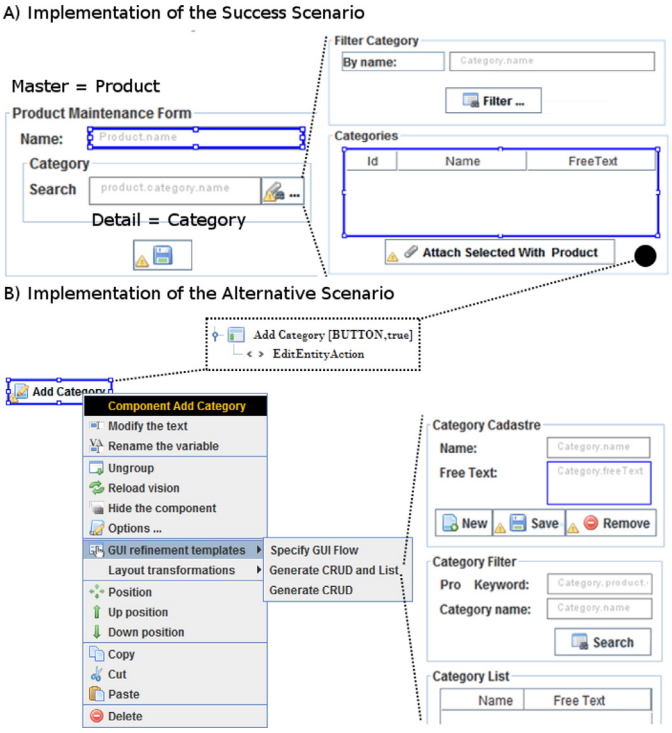

Fig. 16. Implementation of scenarios for the use case `Maintain Products`.

tween *master* and *detail*, entities can be implemented with different structures as well as using different GUI components. Templates for *Detail* are called `refinement templates` and support flexible mockup constructions. Fig. 15 (1) shows the structure generated using a refinement template called `Embedded Form`, Fig. 15 (2) shows the structure generated using another called `Select Single and Attach` and the third is called `Find and Attach`. The original strategy illustrated in Fig. 14 (1), whose template is called `Independent Form`, will persist data from `Product` and `Category` in different database transactions. The strategy used in Fig. 15 (1) owns semantics for actions (i.e., annotations) that dictates that, when the `Save` button is pressed, then the data from the *Detail* = *Category* is persisted in the same transaction as the data from the *Master* = *Product*. The `Select Single and Attach`, shown in Fig. 15 (2), owns semantics that dictates that all the *Details* = *instances of Category* will be loaded from the database, inserted into the combo-box, and the selected one is merged with the *Master* after the button `Save` is pressed. The last, shown Fig. 15 (1), owns semantics that allow the user to specify a filter for the *Detail*, merging the detail into the master. These annotations and the associated UML Profiles are discussed in Section 10.

Success Scenario: Assuming that in the `Paper prototype` the drawing is similar to the strategy shown in Fig. 15 (3), the designer must now detail the actions from end-users derived from this mockup. Thus, the use case `Maintain products` presents the success scenario namely `Find and Select a Category` that must be detailed. Task C is useful for detailing this success scenario for such a use case, as illustrated Fig. 16. This is because through the refinement templates we can assist designers in the modelling of the sequent mockup, designed for the implementation of another associated mockup that *Find* and *Select* the category, required for inclusion in a product. In this example, the mockup shown on the right side of Fig. 16 (A) is automatically generated through a refinement template associated with the button *Find*.

Alternative scenario: Task C is important for detailing alternative scenarios too. The use case `Maintain products` presents an alternative scenario for the case when the category is not found through the the mockup shown on the right side of Fig. 16 (A). Thus, the category must be created so that the end-user re-execute the success scenario. Assuming that the `paper prototype` presents a GUI similar to the one shown on the right side of Fig. 16 (B), the left side of this figure shows the popup menu from MockupToME tool that allows the execution of the appropriate refinement template. This is possible because the designer add a new button on the form for *Filter*. This button is stereotyped as ⟨⟨EditEntityAction⟩⟩, which allows for our tool to associate and recommend refinement templates for detailing the action in a new mockup.

6.2.4. Task D: select the strategy for details

Our tool support is important for the execution of a creative and incremental design process in MDWE, allowing for designers to explore use case scenarios. This is important when some functionalities present uncertainty on the requirements. For example, the case that the client changed his idea about the implementation of a scenario developed in a previous iteration and also along the same iteration. This is a little bit common in start-up contexts (Giardino et al., 2014). In this case, the designer should consider changes before starting the design of models associated with new use cases or the architectural models, discussed in the *Architectural Prototyping Phase*. When this worst-case scenario occurs, then it is important to apply the changes in models, starting by modifications in mockup specifications. Thus, Task D is defined in MockupToME Method for designers to deal with this situation.

Input: Refined mockup (i.e., with different GUI structures).

Output: Refined mockup (i.e., with selected components).

Description: This task is executed only if the designer includes in Task C options for *Master and Detail* that should be re-validated by the *client*, otherwise, the *designer* should skip it and perform Task E. Task D is useful for the worst-case scenario, when textual use cases and the paper prototype present uncertainty from the point of view of *client*. The *designer* may alternate between strategies used to structure each association owned by the *Master* entity. Then, options available for the designed mockups are accorded between client and designer in a second cycle of validation. Besides, considering the worst-case when requirements change with frequency among iterations, thus needing to change models designed in previous iterations. It possible to undo a refinement performed in Task C and also to select which strategy better meets to the requested change in a new iteration.

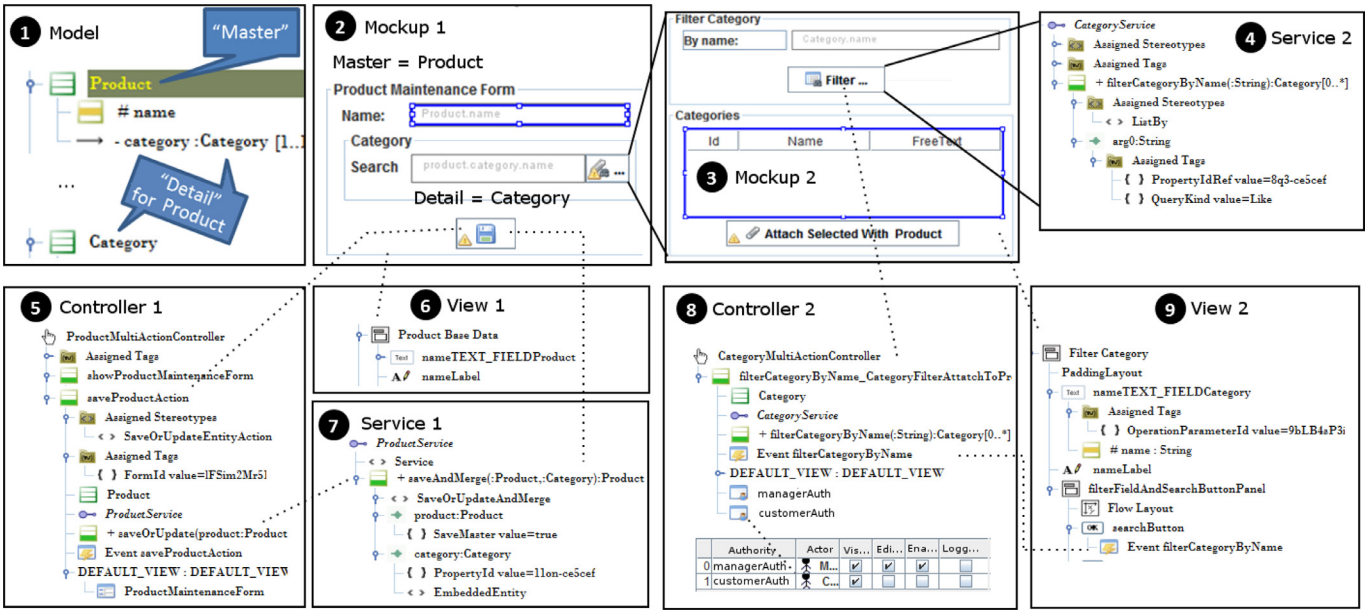

Fig. 17. Design elements represented through the methodology in support for the use case `Maintain Products`.

Exemplification: Tasks C and D allow the execution of an iterative and incremental design approach, always updating previously designed models and keeping them synchronized along the execution of iterations. Consider the mutually exclusive layout strategies shown in Fig. 14 (3). Each menu item will select one of the layouts shown in Fig. 15, which may occur in worst-case scenarios. Tag `LayoutStrategy=xyz` shown in Fig. 15 (1–3) provides semantics that links to the adopted refinement template. Note that, after the selection of a strategy for each *Detail* relationship, mockup components are not removed, but deactivated. These elements allow to group and select strategies for *Master and Detail*. In case a sequent refinement is needed after a selection, than the mockup designer will detail the sequent actions, starting a new instance of the proposed methodology. For example, if the selected strategy is the one illustrated in Fig. 17 (2), the warning icon associated with the button decorated with the *Find* icon suggests that this component needs refinement. In this case, a new mockup annotated with ⟨⟨FilterBy⟩⟩ must be specified and associated with the `Find` button. This is a very similar situation to the use case `List products by preferences`, shown in Fig. 2. MockupToME assists the designer in the representation of the sequential refinements, allowing the automatic generation of new mockup shown in Fig. 17 (3) and the specification of the *Filter* operation shown in Fig. 17 (4) with the help of a wizard.

Client evaluation: Through the popup menu items shown in Fig. 14 (3), and together with the mockup designer, the client can *interact/simulate* with the mockup before the source code is generated, deciding the best structure for a mockup. In the case of non-acceptance or corrections in mockups, previous tasks are executed again until the client decide for a specific structure. In the case of acceptance, the next task is executed. Thus, following the motivating example, assume that the client has selected the option Find and Attach, resulting in an accepted mockup as illustrated in Fig. 14 (2).

Final steps: After client acceptance, GUI components are more detailed, components are standardized in size, position, font, etc.

6.2.5. Task E: generate mockup source code

A choice made by the client about strategies in mockups will allow the mockup designer to generate the source code. This code is used to apply the first test of a runnable prototype (a `Browser prototype` generated directly from a mockup). Thus, associated with the previous tasks, only active mockup components are considered during the source code generation.

Input: Refined mockup (i.e., with different structures).

Output: Browser prototype, accepted mockup.

Description: The execution of Task E implies in the use of a model-to-code transformation that generates source code for HTML 5 directly from mockup. This transformation is simpler than others performed in Task F, which includes model-to-model transformations from mockup to multi-layered model elements named `UML Structural and Behavioural Diagrams`. In this case, only the view layer is generated as source code. In the next phase it is also possible, using model-to-model transformations, to generate what we call `Concrete GUI Components`, characterized by other models in conformity with three other DSLs for GUI (Desktop, Web and Mobile). Both transformations enable the simulation of GUI's flows and user interactions in a web browser. Thus, the client evaluate the `Browser prototype` and, in case of acceptance, the next prototyping phase is executed.

6.3. Part III: architectural prototyping phase

Tasks A to E are used to generate the first compiled prototype based only on mockups. Tasks F to H aim at generating other model specifications that connect GUI DSLs and business layers implemented with MVC-based models. Therefore, instead of using only the MockupToME DSL and entity classes/use cases discussed in the previous phase, the architectural prototyping phase includes model specifications considering heterogeneous DSLs.

In this section, we introduce the underlying architecture that implements the MVC pattern. We separate the business logic from the controller layer to better modularize the source code. Thus, the semantics for business logic is placed in a UML interface stereotyped as ⟨⟨Service⟩⟩. This interface is implemented by other layers such as: (a) Remote; (b) Validation; and, (c) DAO.

The architectural prototyping phase represents the transition from mockup specifications illustrated in Fig. 17 (2 and 3) to other MVC-based layers illustrated in Fig. 17 (4–9) which, follows the structures of a multi-layered MVC. In this phase, models in (4–9) are generated and refined, e.g., detailing properties of GUI components that are not possible to be represented in mockups. In

our first MDWE approach dating back 2008 (Basso et al., 2007) we used to represent these models manually. Due to the introduction of evolutionary prototyping, these models are now automatically generated. Besides, in this stage, the client has already accepted the designed mockups. Thus, this is the correct moment to detail elements associated with the MVC.

6.3.1. Task F: generate MVC layers

In this task, mockups are transformed into MVC-based model application layers as follows.

Input: Accepted mockup.

Output: Concrete GUI components, controllers and services, UML structural and behavioral diagrams.

Description: Once the mockup model is validated and a structure for a mockup is decided, the process towards generating a functional prototype can be executed. This implies in generating all web information systems layers considering the selected domain features. Fig. 17 presents the generation of other layers of the MVC from mockups. Some of these layers are represented with UML Profiles (Entity and Service) and others with DSLs that extends the UML metamodel (Controller and View). The execution of this task outputs the following artefacts:

Concrete GUI components. MockupToME DSL have few properties to set GUI components, thus in a high-level of abstraction than target platforms, e.g., Mobile, Desktop, and Web. To represent a GUI in a target platform, mockups must be transformed by means of specialized DSLs. Thus, three DSLs for GUI can be used and are supported in our set of model-to-model transformations: (1) the Web DSL; (2) the Desktop DSL; and, (3) the Mobile DSL. Fig. 17 (6 and 9) shows the elements generated from mockups (2 and 3) conforms to the Web DSL.

Controllers and services. The buttons specified in mockups own semantics for actions. For example, the `Save` button allows to persist entities and the button `Filter` allows to query a database. These buttons allow us to infer the flow between the user interfaces. For example, MockupToME keeps a trace/flow that connects the mockups shown in Fig. 17 (2 and 3). Another example of inference is the button `Save`, that for the success view will show the `List Form` and for the error view will show the `Crud Form`. Based on these inferences, Controllers (see Fig. 17 - 5 and 8) and Service interfaces (see Fig. 17 7) are automatically generated. Controllers are in conformity with the Action Profile, a DSL that extends the UML we have developed to handle actions commonly associated with the Spring Framework, e.g., simple form controller, multi-action controller, command controller. Service interfaces are in conformity with the Service Profile, a DSL that extends the UML to represent database query semantics.

UML structural and behavioral diagrams. Each application layer derived from the Service or Controller models belongs either to validation, or to persistence, or to remote operation calls, and are generated only when Desktop and Mobile DSLs are used. Fig. 18 shows three layers derived from the interface `ProductService` represented in a UML sequence diagram. This diagram is optionally represented for use cases that are automated through our methodology, because the model, view, controller and service layers are already linked during the transformation from a mockup (see Fig. 17 - 5, 6 and 7). For manually designed functionalities, the messages between these layers must be manually annotated. The exemplified messages define flow and business logic operations related to entity `Product`: (`ProductRemote` or `ProductMultiActionController`) + `ProductServiceValidator` + `ProductServiceDAO Hibernate`.

Generation of platform-specific models for GUI. The GUI components represented in a mockup model are conform to the MockupToME DSL, and must be transformed to one or more platform-specific models for GUI. For example, if requested by the client, the designer must select a mockup model and execute a specific model-to-model transformation to generate a specification in conformity with the Web DSL. Thus, the designer can transform a mockup to one or more DSLs for GUI: `Web DSL`, `Mobile DSL` and `Desktop DSL`. Each platform-specific model for GUI must be manually enriched with details from each DSL. If transformations for desktop and/or mobile platforms are executed, then the generation and refinement of a class from the `Remote Layer` is required (Basso et al., 2014c), which allows to apply remote connections between devices and the web server. Therefore, this approach allows for designers to represent details from each platform supported in the underlying implementation framework.

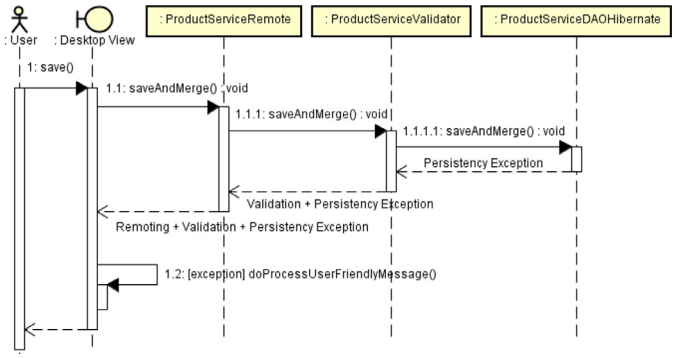

Fig. 18. Message call to process the action `saveAndMerge` in the underlying multi-layered architecture.

Exemplification. For applications that run in a desktop platform, the designer generates a model in conformity with the Desktop DSL, used in the end of our model transformation lifecycle to generate source code mapped into the Java Swing API. The multi-layered architecture allows remote connections from client platform (Desktop) with the server platform. In case of platform-specific models for GUI in conformity with the Mobile DSL, the architecture works in the same way. We have already tested it through remote *http* connections, linking mobile devices programmed with J2ME API and the web server with remote calls, as exemplified in Section 7.3. Thus, the Remote layer delegates operations to a validation layer which is hosted by the web server. In the case of exchange of *View* platforms in the client side, e.g., instead of GUI developed for Desktop use GUIs developed for Mobile devices, at least the validation layer and persistence layer are reused.

Client evaluation. Two model elements generated in this phase are important for client evaluations. **(A)** the `concrete GUI components` which, for the reported experiences in the next sections, uses the Web DSL. The model associated with the artefact `concrete GUI components` is, therefore, a DSL in a platform-dependent model view for GUI in a lower abstraction level than a mockup, which is a platform-independent model for GUI. This model owns a unique structure, does not have deactivated components neither master/detail properties, and its components are able to store specific properties that the mockup does not support, such as events, layout, and appearance properties. **(B)** the controller layer, in which action/event components are defined also as a domain-specific models. Thus, with these two generated elements and considering only the use of Web DSL, a second browser simulation can be performed by clients considering the View and Controller layers.

6.3.2. Task G: detail the business logic

Input: Concrete GUI components, controller, master and detail entities.

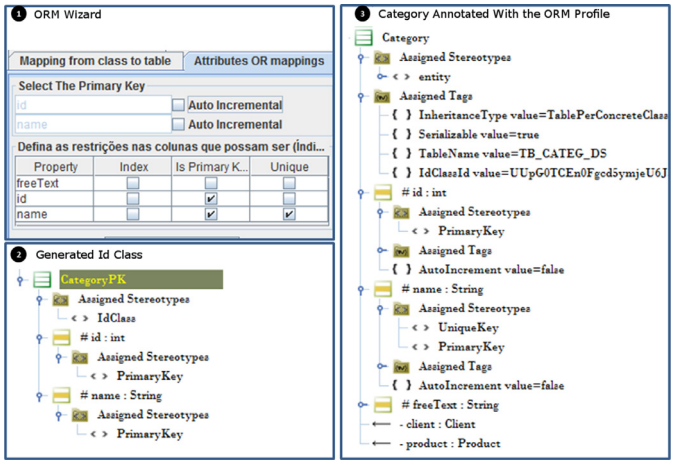

Fig. 19. Entity layer annotated with the `ORM Profile` with the help of a wizard.

Output: Controller and GUI components with authority mappings, service UML interface.

Description: This task intends to generate other specific model layers that are mostly mapped into the DAO layer, to constraint controller layer access, and to constraint GUI fields with security details. The bottom-part of Fig. 17 shows a piece of the wizard to annotate the Controller model responsible to represent access constraints. For an example related with the controller operation, assume that `functionality = elements 4, 8, and 9` from Fig. 17. This example shows that the actor `Manager` has full access to the functionality `Filter Category`, while the actor `Customer` can only visualize it.

6.3.3. Task H: apply UML profiles

This task is assisted by wizards such as the one exemplified in Fig. 19.

Input: Domain-Specific Models (Concrete GUI components and Controller layer), Elements Annotated With UML Profiles (Model layer and Service Interface).

Output: Input elements with more annotations to allow the execution of platform-independent model to platform-specific model transformations.

Description: This task is optional, since one can be interested in transform domain-specific input models into UML models. Besides, aiming at generating a more complete source code, the *designer* can specify some details such as annotations, not generated by previous transformations. To represent annotations for ORM, it is used a wizard to decorate entities. This is exemplified in Fig. 19 (1), where a wizard allows the generation of an `Id Class` in Fig. 19 (2) followed by a guided annotation, resulting in the annotated entity named `Category` as illustrated in Fig. 19 (3).

Source code generation: Model-to-code transformations are applied against the input elements to map them to the Java architecture used by the development team. This transformation enables the generation of a functional prototype, since all layers are generated as source code. Afterwards, source code is refined by programmers and then tested. For example, ORM annotations are used to generate Java classes decorated with the JPA (Burke and Monson-Haefel, 2006), as exemplified in Section 7.

Client evaluation: Finally, the client performs his/her forth interaction for the acceptance test. Then, improvements and corrections are made in the generated functional prototype, delivering a working piece of application, the last software artefact as shown in Figu. 11.

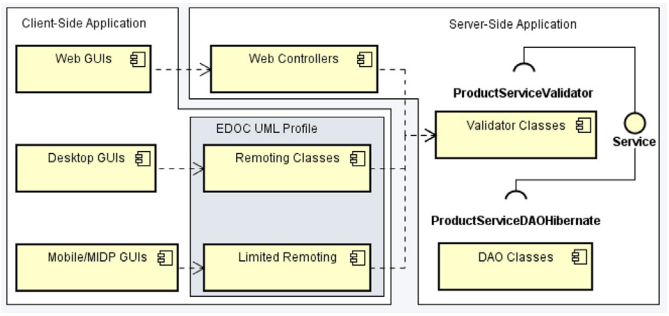

Fig. 20. Component diagram describing the architecture.

6.4. Part IV: functional prototyping phase

A fully executable piece of software is obtained in this phase, where generated prototypes are target for acceptance tests. These artefacts are then detailed and used by a development team. Using model-to-code transformations it is possible to generate functional prototypes, which can be tested by clients in web browsers. Therefore, a functional prototype is a fully implemented functionality, e.g., considering the implementation of database transactions and queries, which must be tested by clients in a real-world scenario.

6.4.1. Task I: generate complete source code

Input: All aforementioned models.

Output: Source code for MVC-based layers.

Description: The result is a fully testable platform-independent model prototype. This is achieved after the usage of a platform-independent model to generate platform-specific model transformations. This means that all strategies used in annotated mockups imply on the use of different transformations from platform-independent models to platform-specific models. Currently, model transformations enable the generation of source code for the following layers:

1. Model-Entity layer with support of object-relational mapping details.
2. Controller-Business layer with support for transactions involving the service/business layer and calls for a remote access layer.
3. Controller-Persistence layer with the layer for handling the data access object.
4. Controller-Actions layer to handle GUI events.
5. View layer in the client side application.

Fig. 20 illustrates components that, except for the *Model-Entity*, implement aforementioned layers of a functional prototype. Thus, as part important of the Functional Prototyping phase, Section 7 provides information about the implementation of these components.

6.5. Final remarks

It is important to mention that the usage of model transformers to refine mockups is a practice that can also be used by other MDWE proposals. Thus, the concepts introduced in MockupToME for semi-automated refinement of mockups are general and useful for researchers and practitioners of MDWE.

7. Implementation

The aforementioned automated design tasks were used in the development of real-world web information systems. In this section, we present implementation details, including artefacts generated in the *Functional Prototyping Phase*.

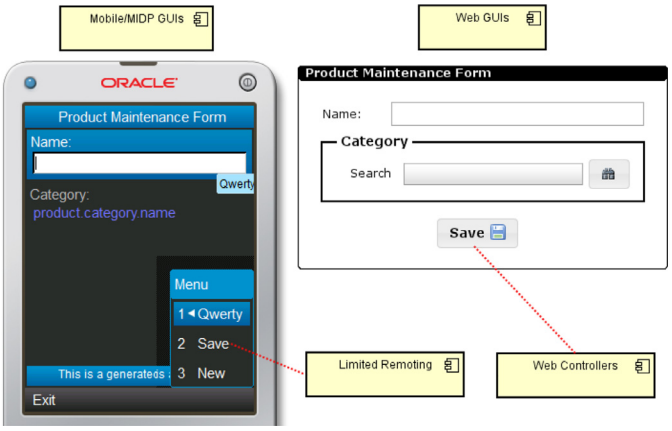

Fig. 21. Resultant GUIs executed in a mobile emulator and in a web browser.

7.1. Underlying architecture

Following the motivating example, this section exemplifies the classes that are generated from the models represented in the *Architectural Prototyping Phase*. These classes are connected in a logic flow illustrated in Fig. 18 as a UML sequence diagram. For GUIs developed for Desktop or Web platforms, these method calls are similar. Both GUIs will delegate the processing of business rules for a Validator class hosted in the web server which, in turn, delegates the task of persistence for the persistence layer. This flow between layers is performed/injected by Spring Framework. When one need to change something in the application regarding the business layer, it will be made in the control layer `xxxMultiActionController`, and/or in the validation layer `xxxServiceValidation`, and/or in the persistence layer `xxxServideDAOHibernate`, where `xxx` is the name of the associated entity. Actions found in screen flows will be handled in the controller, or in action listeners developed for Swing or in commands developed for J2ME/MIDP.

The generated source code includes the GUI layer for Mobile, Desktop and Web platforms, data access layer, entity layer, integration/remote layer, xml configuration files, text files, Java classes, data base scripts, models, etc. The following configuration of technologies from the underlying target platform are used in WCTSample, the pre-configured MVC framework for web development used in our experiences, which is composed by: Hibernate, JPA, jQuery, JSTL, Swing, PostgreSql, Apache Commons Validation, and Spring Framework. This architecture is flexible and supports changes through the FOMDA Approach (Basso et al., 2013). For example, the following technologies were changed in the underlying implementation of WCTSample across software projects: (1) first, software projects dating back 2008 used HBM files to apply Hibernate mappings (ORM) and in recent projects JPA was used; (2) the first software project was developed at Adapit adopting Dojo-toolkit API as web technology to write rich GUIs, and in the second project we used jQuery.

7.2. Generated source code

Fig. 21 shows the resultant GUI from the overall methodology. This GUI is executed in a web browser and represents the functional prototype for the use case `Maintain products`. Behind this simple GUI, several application layers based on the MVC connects GUI components, action and flow handling, field validation and database persistence. These layers are presented in the following.

The source code generated for the entity `Product` is illustrated at the center of Fig. 22. Note that a dashed line includes the mapping from tags and stereotypes from our UML Profile for ORM to the JPA representation. Besides, XDoclet comments such as the operation `getName` are also mapped into the Apache Commons Validator API, which automatically validate GUI form fields. As long as the MVC designer specifies ORM annotations using the wizard shown in the right side of Fig. 22, the source code generated for Entities will not require manual changes.

Fig. 23 exemplifies the source code generated for the Validation layer located in the server-side. Each action/button specified in a mockup that semantically submits a form, e.g., ⟨⟨SaveOrUpdate⟩⟩ and ⟨⟨FilterBy⟩⟩, also presents an implemented operation into the `xxxValidator` class. The implementation of the operation `saveAndMerge` delegates for the Spring Framework API the checking if the data from the instance of `Product` are valid. In the case it is valid, then the operation delegates the task to persist the instances of `Product` and `Category` to the DAO Layer (injected into the property `productService`). In case of invalidity, then an exception is thrown to be handled in the client-side, where a GUI will presents a user friendly message. The developer is free to include a specific validation in source code if he/she needs. Thus, this operation do not require changes to work in a functional prototype.

Fig. 24 shows the implementation of the DAO layer with the Hibernate. Soon after opening a database transaction, the instance of entity `Category` is set to a persistent state: `session.refresh(category)`. This clean any information owned into the parameter `category` except the primary key. This is due to the stereotype ⟨⟨EmbeddedEntity⟩⟩ assigned by the parameter illustrated in the top of Fig. 24, automatically generated along the transformation from a mockup to the Service UML Interface. This operation do not require changes to work in a functional prototype.

Fig. 25 shows the source code required to handle the action `saveProductAction`. This action is mapped to the button `Save` from the JSP source code presented in Fig. 26. The operation `saveProductAction` first binds the request parameters into the properties of the entities represented in the mockup, then it propagates the validation and persistence for the other layers. The last part of the source code is dedicated to process exceptions that came from the *Validator* and *DAO* layers. Note that the validation is delegated to the *Validator* layer, which is injected into the property `productService`. This controller has a considerable amount of source code because it was generated based on the Multi-Action controller from the Spring Framework, which have several operations.

The source code for JSP (Burke and Monson-Haefel, 2006) is shown in Fig. 26. The top-part of Fig. 26 (A) shows the header information included in all root JSPs, e.g., the ones directly associated with a mockup, with the information necessary to use the access control functionality from the WCTSample framework. The bottom part of Fig. 26 (A) shows the source code that maps the properties from entity `Product`, e.g., `id` and `name`. Fig. 26 (B) shows the source code for pane `Category` which associates the action of button `Find` to the controller `CategoryMultiActionController`.

7.3. Implementation for mobile

These source code illustrate the minimum artefacts generated by our approach in the development of a functionality for web information systems. However, other devices can connect with these functionalities available on the web server. For instance, the web server can provide access to external components, such as GUIs developed to run in mobile devices and in desktop applications.

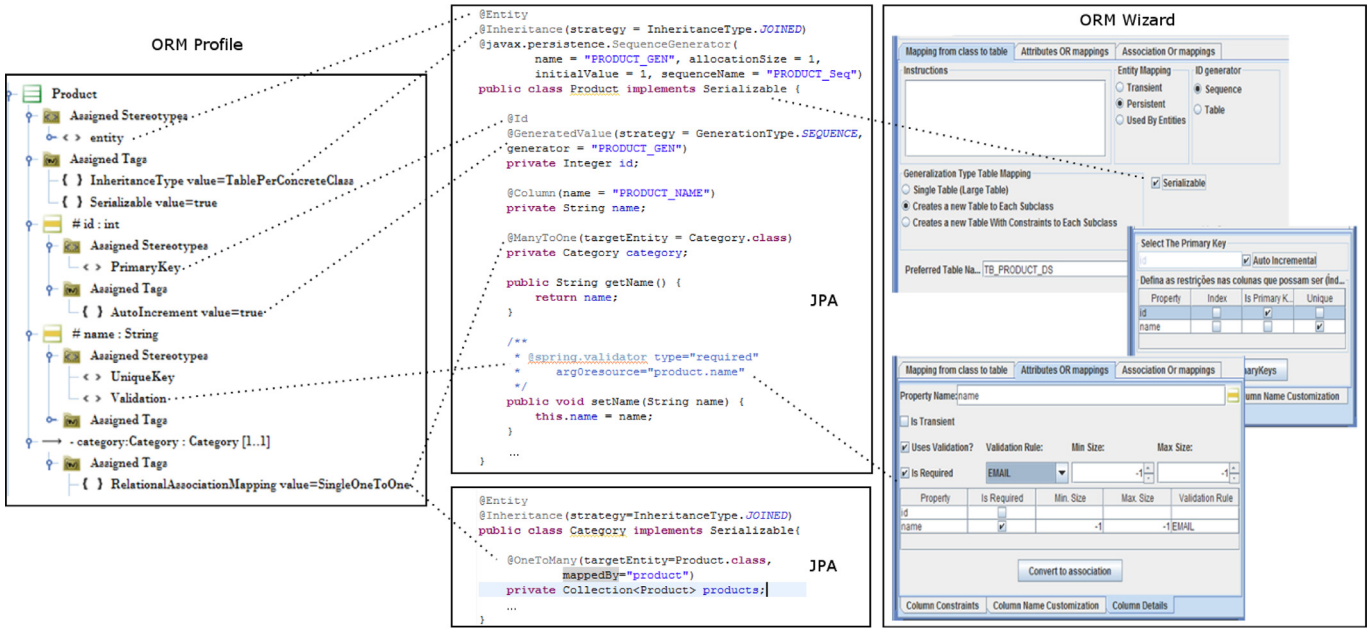

Fig. 22. Generated source code for the *Model* layer implemented with JPA (2.0).

```
/**
 * @spring.bean id="productServiceValidator" singleton="true"
 */
public class ProductServiceValidator implements ProductService {

    private ProductService productService;
    private BeanValidator validator;
    /**
    *@spring.property ref="productServiceDAOHibernate" singleton="true"
    */
    public ProductService getProductService(){
        return productService;
    }
    @Override
    public Product saveAndMerge(Product product, Category category)
       throws FieldMsgValidationException, ValidationException,
             NonUniqueObjectException, ConstraintViolationException,
             DataException, Exception{
            //Automatic validation with Apache 'commons validator'
            BindException errors1 = new BindException(product, "product");
            validator.validate(product, errors1);
            //in case product data is not valid, throws a ValidationException
            //for custom validations, then throws FieldMsgValidationException
            if (errors1.hasErrors()) {
                throw errors1;
            }
            //case is valid, call the DAO layer to persist product and category
            return this.productService.saveAndMerge(product,category);
    }
    //...
}
```

Fig. 23. Generated source code for the *Validation* layer implemented with Springframework and Commons Validator.

Thus, our model transformation lifecycle supports the generation of GUIs programmed with J2ME and, for desktop applications, programmed with J2SE. Thus, two other DSLs are important besides `Web` DSL: the Mobile and Desktop DSLs.

We demonstrate the worst-case scenario on the refinement of a model in conformity with Mobile DSL. The connection between *MockupToME DSL* and *Mobile DSL* is through transformations, as illustrates Fig. 27. The first DSL holds richer types of GUI components than those available in the Mobile DSL. If a GUI for mobile platform is needed, then the designer executes a transformation from a `model 1`, which is in conformity with MockupToME DSL, to another `model 2`, which in conformity with the Mobile DSL. This transformation is illustrated in Fig. 27 (1) and shows a piece of a model-to-model transformation. The result is the `model 2` shown in Fig. 27 (2).

```
/**
 * @spring.bean id="productServiceDAOHibernate" singleton="true"
 *  */
public class ProductServiceDAOHibernate
    extends HibernateDaoSuport implements ProductService {

    @Autowired
    private SessionFactory sessionFactory;

    public List listByCategory(String categoryName)
            throws DataAccessException {
        String name2 = "%" + categoryName + "%";
        return openSession().createQuery(
                "from Product product " +
                "where product.ctegory.name like ?")
                .setParameter(0, name2).list();
    }

    public Product saveAndMerge(Product product, Category category)
       throws FieldMsgValidationException, ValidationException,
             NonUniqueObjectException,
             ConstraintViolationException,
             DataException, Exception{
        org.hibernate.Session session = openSession();
        try{
            session.beginTransaction();
            session.refresh(category);
            product.setCategory(category);
            session.saveOrUpdate(product);
            session.merge(category);
            session.merge(product);
            session.getTransaction().commit();
            return product;
        }catch(Exception ex){
            ex.printStackTrace();
            session.getTransaction().rollback();
            throw ex;
        }finally{
            if(session != null && session.isOpen()) session.close();
        }
    }
    //...
```

Fig. 24. Generated source code for the *DAO* layer implemented with Hibernate in HQL.

Mobile DSL represents the J2ME profile named CLDC, thus for a limited GUI API. Because `model 2` owns less representative GUI components than the *Mobile DSL*, there is lost of information about layout from transformation from `model 1` to `model 2`. For ex-

```
@Controller
public class ProductMultiActionController
    extends MultiActionController{

    @Resource(name="productServiceValidator")
    private ProductService productService;

    public Product bindProduct(HttpServletRequest request)
            throws Exception{
        Product product = new Product();
        product.setId(java.lang.Integer.parseInt(
                request.getParameter("product.id")));
        product.setName(request.getParameter("product.name"));
        return product;
    }

    public Category bindCategory(HttpServletRequest request)
            throws Exception{
        Category category = new Category();
        category.setId(java.lang.Integer.parseInt(
                request.getParameter("product.category.id")));
        return category;
    }

    @RequestMapping("/saveProductAction.html")
    public ModelAndView saveProductAction(
            HttpServletRequest request,
            HttpServletResponse response ){
        try{
            //bind form attributes into the entities
            Product product=bindProduct(request);
            Category category=bindCategory(request);
            product.setCategory(category);
            //call the validation to process the action saveAndMerge
            productService.saveAndMerge(product,category);
            //Handle the user friendly message
            DialogKind kind = DialogKind.SucessDialog;
            request.setAttribute("msg", "SucessDialogMessage");
            request.setAttribute("kind", kind);
            request.setAttribute("title", "SucessDialogTitle");
            //In success, show the GUI that contains <<FormList>>
            return showListAllProductsView(request, response);
        }   catch(FieldMsgValidationException ex1){
            doProcessUserFriendlyMessage(request,ex1);
        }   catch(ValidationException ex2){
```

Fig. 25. Generated source code for the *Controller* layer implemented with Springframework (2.5).

A) Content of the Master "Product" in JSP Source Code

```
<%
UsuarioDTO userDTO =(UsuarioDTO) session.getAttribute("user");
LocalUserService localUserService = LocalUserService.getInstance();
%>
<%  if("null".equals(request.getSession().getAttribute("login"))
     || null == request.getSession().getAttribute("login")){%>
        <script>
            jqueryOpen('showLoginInformationForm.html','general');
        </script>
<%  }%>
<%@page import="entities.Product"%>
<% Product product = (Product) request.getAttribute("product"); %>
<% if(product == null) product = new Product();%>
<%@page import="entities.Category"%>
<% Category category = (Category)request.getAttribute("category");%>
<form class="default_form" method="post" id="productForm">
    <input name="product.id" id="product.id" type="hidden"
        value="<jstl:out value='${product.id}'/>"/>
    <input name="product.name" id="product.name"
        value="<jstl:out value='${product.name}'/>"/>
```

B) Content of the Strategy Find and Attach in Association with the Detail "Category"

```
<div id="findPanel">
    <span class="ui-ti-text"><binder:message code="Category"/></span>
    <input name="product.category.id" id="product.category.id"
        type="hidden"
        value="<jstl:out value='${product.category.id}'/>"/>
   <ul> <li>
<%if(product!=null && product.getCategory()!=null
    && product.getCategory().getId()!=null
      && product.getCategory().getName()!=null){ %>
        <label for="category.find">Find:</label>
        <input type="text" id="category.find" name="category.find"
            value="<jstl:out value='${product.category.name}'/>"/>
  <%} %>
        <button onClick="jquerydoPost('productForm',
        'findCategoryAction.html?product.id=<%=product.getId() %>',
            'content'); return false;" >
        <span class="ui-button-text">
        <img src="imgs/find.png"/>
        </span>
```

Fig. 26. Generated source code for the *View* layer in JSP.

ample, the transformation illustrated in Fig. 27 (1) show that a component instance of `ScreenLayoutSpecification` (in conformity with MockupToME DSL) is transformed to another component instance of `Form`, which is in conformity with Mobile DSL for CLDC. While the first supports layout managers such as a flow layout, used to centralize components such as the button `Save` illustrated in Fig. 21, the second does not support layout manager. This component in `model 1` in conformity with MockupToME DSL is of type `Button`, while for the model 2 it is instance of `Command` in conformity with Mobile DSL, as illustrates Fig. 27 (4).

The best-case scenario for mobile is also supported. Thus, although the information about the flow layout is lost in the second model, this is a constraint from the Mobile platform, not a limitation in the MockupToME Method. For example, for richer GUI components that can run in other mobile devices, it is possible to include in the model transformation lifecycle another package for Mobile DSL. This package allows the designer to represent GUI components mapped for the CDC profile, which allows the representation of GUIs programmed with Java AWT. Likewise, for desktop platform we include the Desktop DSL, which is mapped for source code developed with Java Swing. For CDC platforms, a transformation from MockupToME DSL to Mobile DSL will not imply in lost of layout information.

For the design of Mobile GUIs, we prefer the use of Matisse Designer[5], as shown in Fig. 27 (4), instead of the EMF editor shown in Fig. 27 (2). This means that, differently from the other DSLs included in our lifecycle, the refinement of a GUI for mobile is performed with an external tool. Because Matisse exports and imports the design in XML, this external tool is integrated with our supporting tool through operations of type import/export. This means that the `model 2` is transformed for the XML in conformity with the Matisse Designer, resulting in `model 2'`. As illustrates the piece of source code in Fig. 27 (3), we also developed a model-based operation of type text-to-model that reverses data from XML (`model 2'`) to the representation in `model 2`, which is in conformity with `Mobile DSL`.

The limitation in user interactions from the CLDC profile have no effect the model elements represented on the web server. However, as illustrated in Fig. 20, for establishing the connection of the mobile device with the business logic available on the server side, it is needed to use an API that deals with limited remote connections in the client-side of the application. The design and implementation of such connection are discussed in the next subsection.

7.4. Remote connection

As illustrates the component diagram in Fig. 20, devices on the client-side are connected with the business logic available on the server-side through remote connections. Likewise, any DSL added for the *View* layer, i.e., which will not executes inside the web server, demands a new layer in the MVC structure to handle the remote connection between the device and the web server. This allows that the whole source code for application logic remains isolated on the web server, a good approach for modularity and maintenance of source code (Allier et al., 2015).

As illustrated in the top-part of Fig. 28, a class named `RemoteProductService` is mapped for the *Remote* layer, thus

[5] Matisse Designer - <https://netbeans.org/community/magazine/html/03/matisse/ >

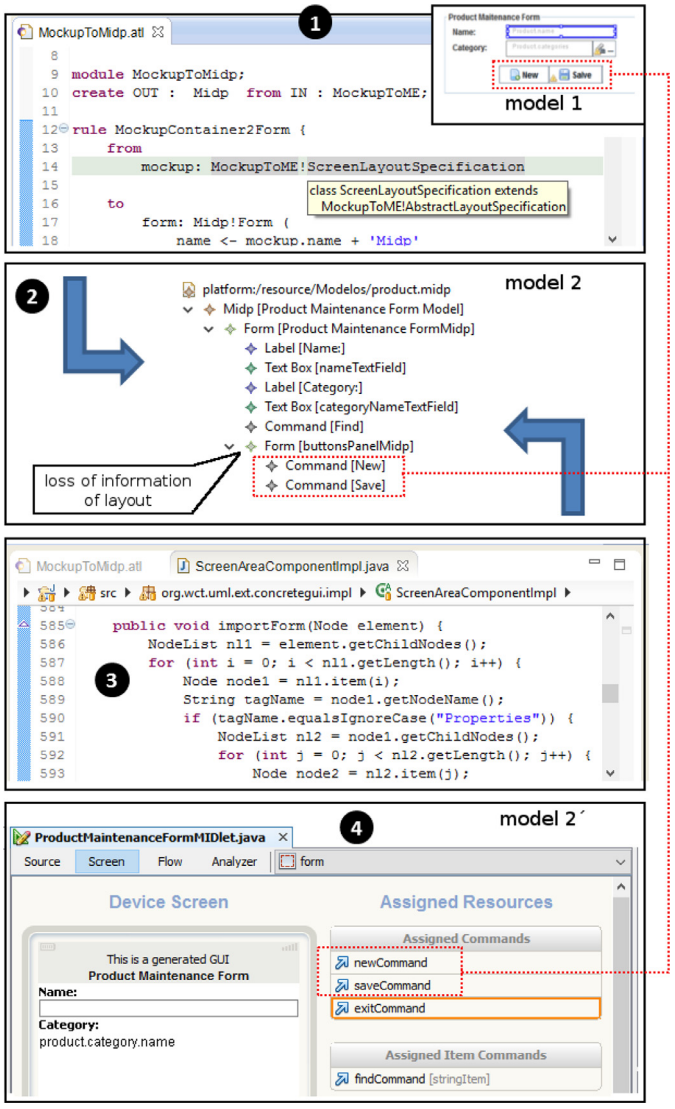

Fig. 27. Generated models for the *View* layer on a mobile device dependent from the CLDC platform: the source code in (1) allows the generation of a model conforms to *Mobile DSL* shown in (2) and; the source code in (3) allows to import/export such model for the XML file associated with the Matisse (Mobile Designer) shown in (4).

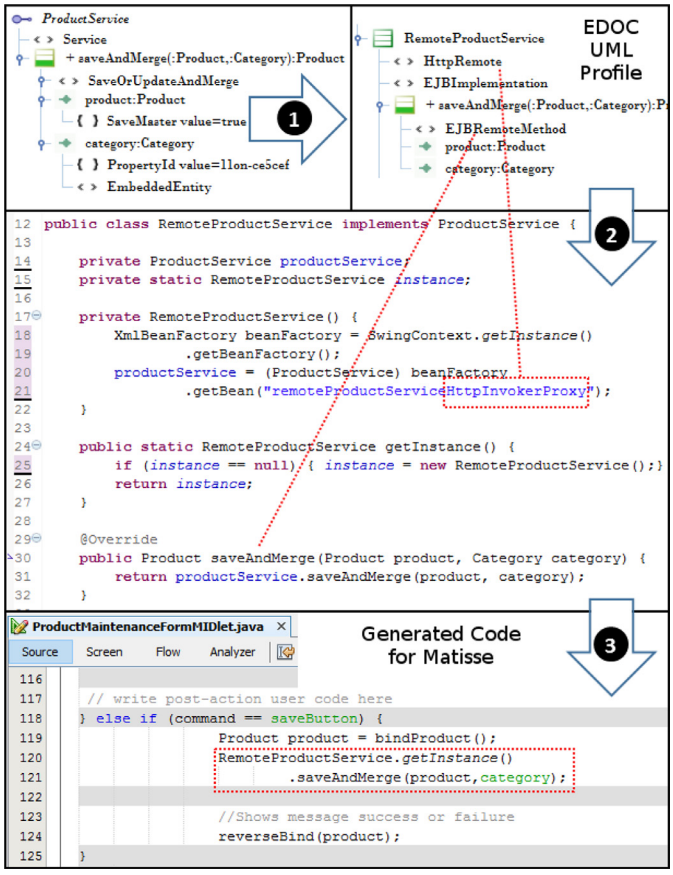

Fig. 28. Generated source code for the *Remote* layer in HTTP Remote.

a piece of a multi-layered MVC structure. This new class is generated from the representation of the `ProductService` interface. It holds semantics for connection in conformity with annotations from the EDOC UML Profile, such as ⟨⟨EJBImplementation⟩⟩ and ⟨⟨EJBRemoteMethod⟩⟩. Such specification is transformed into the source code illustrated in Fig. 28 (2). The class `RemoteProductService` is also considered for the generation of the source code shown in Fig. 28 (3), which is used by the Matisse plug-in for simulation of GUIs for mobile devices.

7.5. Final remarks

The example shown in Fig. 27 illustrates the flexibility promoted by a multi-layered architecture. In case the designer needs to include a representation for an external mobile GUI, such as a DSL mapped to Android SDK[6], it would be enough to: (1) develop a model-to-model transformation from `model 1` in conformity with MockupToME DSL into a `model 2` in conformity with the hypothetical Android DSL; and (2) develop a model-to-code transformation from `model 2` to the underlying implementation associated with this API.

Each inclusion of a DSL in the lifecycle of model transformation implies in a set of new models and refinements. This is because MockupToME DSL is defined in a high-level of abstraction than these platform specific DSLs for GUI. The inclusion of the hypothetical Android DSL would not imply in change for the *Evolutionary Prototyping Phase*. However, it would imply in inclusion of such new DSL into the *Architectural Prototyping Phase*. For example, in this phase the use of CDC requires a different package from the Mobile DSL and also a different model-to-model transformation in the lifecycle illustrated in Fig. 27.

Through the FOMDA Approach (Basso et al., 2013) we add flexibility for the methodology in the implementation level. Although we consider that this approach for MDE as Service is not easy neither cheap, it is very flexible to include new DSLs and model transformations as requested by software factories. Thus, it is possible to evolve the presented methodology in terms of new DSLs, new MVC layers, underlying implementation frameworks and APIs. The generation of source code, strictly from what is designed, is possible due to the development of some DSLs and the integration of several concepts considered in the literature of the area. Other DSLs are not included in our lifecycle for model transformations and methodology, which means that the reader should consider this as a limitation of our proposal. Despite this limitation, we provided an interesting set of tools in support for automated design of web information systems.

[6] Android SDK - <http://developer.android.com >

```
143    @RequestMapping("/generateBilletAction.html")
144    public ModelAndView generateBilletAction(
145                              HttpServletRequest request,
146                              HttpServletResponse response) {
147        try {
148            JBoletoBean jBoletoBean = new JBoletoBean();
149            ShoppingKart sk = new ShoppingKart();
150            sk.setId(request.getParameter("id_shoppingkart"));
151
152            List<Product> products = productService
153                    .listProductsByShoppingKartId(sk);
154
155            Vector<String> items = new Vector<String>();
156            float total = 0.0f;
157            for (Product prod : products) {
158                String item = "Item:" + prod.getName() + "-"
159                        + prod.getCategory().getName() + " R$ "
160                        + prod.getValue();
161                items.add(item);
162                total += prod.getValue();
163            }
164            jBoletoBean.setDescricoes(items);
165            jBoletoBean.setValorBoleto(total);
166            jBoletoBean.setDataVencimento(NEXT_MONTH);
167            return generatePDF(jBoletoBean);
```

Fig. 29. Manual implementation of `Generate Billet` in the controller layer.

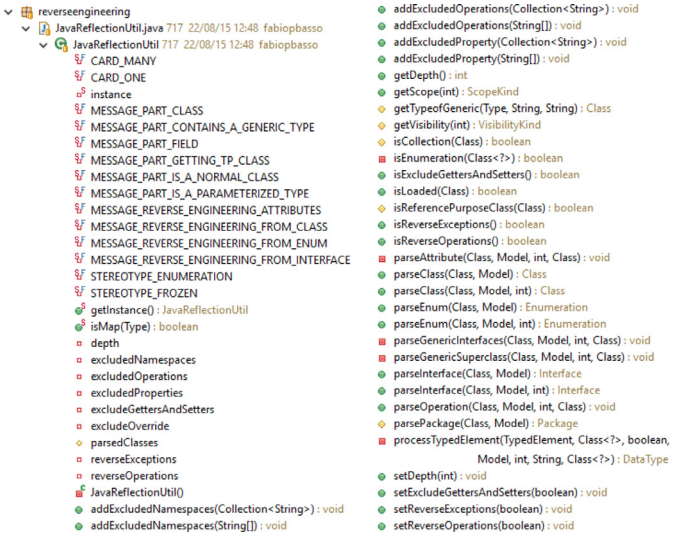

Fig. 30. The class developed for automatic reverse engineering.

8. After source code generation

This section describes activities recommended to execute after the functional prototyping phase.

8.1. Round-trip engineering

Full source code generation is achieved in our experiences. By full we mean from the perspective of what is designed, not the overall application. Thus, the full generation is for few use case patterns. This practice have a drawback: for non generated functionalities, which are programmed without the use of model transformations, round-trip engineering can be necessary. This would imply in an overhead to synchronize source code and models. For example, when functionalities developed without MockupToME require the developers to change manually an already generated source code. Round-trip engineering is necessary to ensure that regenerated source code is correct.

To exemplify how this issue is tackled in our approach, consider that, after the source code generation discussed previously, the use case `Generate Billet` has to be manually developed. We acknowledge that such use case is possible to be abstracted in a model, demanding only an increment in a DSL discussed previously. However, assume that this is not the case and the developer must develop it manually.

In the worst-case scenario, the implementation of the controller layer follows as is illustrated in Fig. 29. It is composed of the following other implementations: (1) assume that the developer has added the operation `generateBilletAction` inside the previously generated class shown in Fig. 25, representing the controller layer; (2) line 152 shows a call for the DAO layer illustrated in Fig. 24, which contains a manually implemented operation to list products from a web shop kart; (3) line 167 shows an operation that generates the PDF file with information for payment, also developed manually inside the controller.

In this case, two previously generated classes where changed manually, requiring the execution of a manual round-trip engineering. As consequence, the developer or designer needs to update the model elements (5) and (7) shown in Fig. 17. They need to specify manually one operation in each model element with a tag *BodyCode*, whose internal source code is represented as a string. This is an issue when developers or designers are inexperienced. Thus, whenever possible, this reverse and manual round-trip engineering should be avoided.

Our recommendation to mitigate the need for reverse manual round-trip is simple and also well-known in the literature (Kelly and Tolvanen, 2008; Whittle et al., 2013). Developers should not develop new features inside the generated source code, thus developing new classes and, preferably, locate it in a separate package. Instead of adopting an approach that leads to the worst-case scenario, we recommend that the developers: (1) create a new package; (2) manually develop another class for controller, e.g., `ManualCodeMultiActionProductController`; and, (3) manually develop another class for DAO, e.g., `ManualCodeProductDAOHibernate`.

These recommendations will keep isolated the generated classes from the manually coded ones, reducing the chances for the execution of manual round-trip engineering from code to model. The generation of source code must obey the same rules. Of course, there are always exceptions to these rules. In our experiences, we observed that entities, in the *Model* layer, are the unique source codes were conflicts between manual and generated source code occurs, independently from the aforementioned recommendation. This is because entities are the centralizer source code in a DDD approach. While other layers accept the development of several classes, the *Model* layer is more rigid. For example, we can develop two classes for the DAO layer, namely `ProductDAOHibernate` and `ManualCodeProductDAOHibernate`, but they both refer a unique entity class `Product`.

Changes in entities require round-trip engineering. However, this is not a big deal. We are not giving too much credit in regard to process overhead when it is necessary to apply specific changes in entity classes, e.g., into the model element shown in the left side of Fig. 22. That is because the wizard illustrated in the right side of Fig. 22 helps in the execution of these manual round-trips. Moreover, as illustrates Fig. 30, we also developed a feature for automatically reverse Java source code to model. This feature is part of our tool support integrated with the Java platform. It is connected with adaptive test cases and model transformations, allowing the execution of operations of type code-to-model (Basso et al., 2014a).

It is important to mention that our tool support for automatic reverse round-trip is limited and it is not applicable to the View layer (JSP source code). For the worst-case scenario, whose changes

are not supported by wizards neither by automatic reverse engineering, the software engineer must plan whether the execution of a manual round-trip engineering is needed and when it should be made. Manual round-trip engineering is not a mandatory task for each iteration. For example, the software engineer should plan whether its execution takes place in the current iteration or in the next or only in the end of the project (Basso et al., 2015). This decision depends on the ability of the teams to perform manual round-trip.

For those frequent changes observed along iterations, another possibility to avoid manual round-trip is to introduce them into model transformations. Likewise, our tool allows extensions in model transformation components, made on the fly, through specialization points (Basso et al., 2014c). Preferable, these extensions should be added before the beginning of software project, when these components are adapted in the pre-game phase (Basso et al., 2015). However, they can also be introduced along the execution of a software project. To reach benefits and drawbacks that this approach introduces for a software project, a future work will discuss how a team received increments in components for source code generation along the execution of iterations.

It is also good to remember that a version control system helps developers to compare versions of the generated source code. Developers trace overwritten artefacts in a new source code generation (forward round-trip), making punctual manual adjustments in source code when needed. Artefact that is not fully generated implies on a big effort for developers to apply adjustments. We are able to generate full source code, which mean that, when source code generators are calibrated and without errors, developers do not spent a big effort in making adjustments. Therefore, even considering the worst-case scenario, developers are capable to make punctual adjustments in source code, which is also not considered in our experiences as a big issue that hampers the execution of iterations.

8.2. Acceptance tests

From the generation of functional prototypes, acceptance tests are conducted in iteration cycles following the selected reference model for software development process. Thus, it is important to test in a web browser or in a mobile emulator, together with the client, each use case scenario developed or not with the assistance of our tool support.

For a general guidance on the execution of acceptance tests, we found very interesting the Acceptance Test-Driven Development (ATDD) (Gärtner, 2012). It includes some practices for the execution of acceptance test cases, which can be performed manually or automatically with the framework JBehave[7]. After the development of the new use cases, automated test cases in JBehave can quickly detect whether functionalities, developed in previous iterations, fails in the new iteration due to the introduction of new source code. For this reason, acceptance tests are complementary to the tests/validations with clients performed in the Evolutionary and Architectural Prototyping Phases, which consider only features developed in the current iteration.

We have no clear position about whether these tests should be developed in conditions of worst-case scenarios, when requirements changes with many frequency such as in start-ups (Giardino et al., 2014).

8.3. Final remarks

Worst-case scenarios, such as those found in start-ups, need the execution of iterations considering features for innovation (Giardino et al., 2014). Authors state that this can imply in requirements that change very fast. This is an issue when performing time-scales planned for one month or more (Whittle et al., 2013). Thus, other authors suggested the execution of tasks for discovery and invention (Schwaber, 1995), which is a characteristic from rapid application prototyping few understood in research and practice of MDWE.

For these reasons, we recommend the execution of Tasks C-E in the MockupToME Method. In our approach, designers are free to explore alternatives for implementation of one or more use cases/user stories. We believe that this possibility influences positively the use of activities performed after the source code generation. In this point, we have some lessons and open questions, as discussed in the next section.

9. Experience report

This section reports on an industrial innovation effort developed in collaboration with the Brazilian start-up company Adapit. Five applications were developed using the automated design approach for MDWE (integrally or partially) to the following domains: agribusiness management, online auction, trainee management, quality management, and financial management. We selected two software projects, presenting a summary about the application of these techniques in each one. Table 1 presents the team configuration associated with the design and source code generation tasks, where "SP^i is a software project for i in 1-2":

SP^1-ERP and CRM for online auction. This is a web/desktop application to apply online auction with support for Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM). It was developed between 2007 and 2008 by Adapit using AMDA as framework for software process (Basso and Oliveira, 2007). The team configuration that was allocated to this project is shown in Table 1 and includes a developer and a designer. SP^1 was conducted only with the Architectural Prototyping and Functional Prototyping phases because at the time we had not developed the MockupToME nor tasks associated with Evolutionary Prototyping Phase.

SP^2-ERP for financial management. It is a web application to manage financial support for innovation projects. The latter application has been developed between 2010 and 2011 by Company A. This project was supervised by the Adapit's team. As shown in Table 1, to perform the activities discussed previously, Company A allocated in SP^2 a team with low design and development experience. The team in SP^2 adopted integrally the tasks and tool support discussed in this paper.

In the following we discuss: (1) the experience that justifies the advent of MockupToME Method and tool support; (2) discrepancies of these software projects that suggest improvements from our current approach in comparison to our previous initiative; (3) lessons learnt from these projects; and, (4) open questions derived from these experiences.

9.1. Justification

In 2007, the development of wizards in support to MDWE was part of services added in a business plan proposed by Adapit, incubated at RAIAR-TECNOPUC-Brazil between 2007 and 2011. The execution of SP^1 was far to be considered as productive, leading to the conclusion that this business plan failed due to difficulties to execute our first approach for MDWE in start-up contexts. For example, in SP^1 we observed issues associated with round-trip engineering, which leaded to some good practices discussed previously. Our worst-case scenario in SP^1 occurred when we performed an iteration lasting two months. After five weeks, the client changed his idea about the requirement in the first cycle of validation.

[7] JBehave - <http://jbehave.org/ >

Table 1
Attributes of the teams used in each software project.

Attributes of the team used in SP[1]							
Num.	Stakeholder	Experience	Education level	Exp. in UML	Exp. in MDE	Exp. in J2EE	Exp. in MVC
1	Designer	Senior	Master degree	Advanced	Advanced	Basic	Advanced
1	Programmer	Senior	Master degree	Advanced	Advanced	Advanced	Advanced
Attributes of the team used in SP[2]							
Num.	Stakeholder	Experience	Educationlevel	Exp. in UML	Exp. in MDE	Exp. in J2EE	Exp. in MVC
1	Designer	Trainee	Graduate	Basic	None	Basic	Basic
1	Programmer	Trainee	Undergraduate	Basic	None	Basic	Basic

This required the re-execution of the phases Architectural Prototyping and Functional Prototyping, resulting in changes in source code and model. Because we had no tool support for reverse engineering, this experience in SP[1] suggested that round-trip engineering should be executed manually. It was executed before restart the Architectural Prototyping phase, re-generating source code in Functional Prototyping phase and making manual adjustments in source code.

We concluded that manual design of MVC-based application models is very tiring and expensive to the point that, in 2008, Adapit considered the possibility to leave the MDWE approach. We agreed that MDWE was not productive for the company context: start-up and small company, with few money and new in the market. However, it was decided to give for MDWE one more chance. Likewise, issues observed in SP[1] leaded to the development of the MockupToME Method. This allowed professionals from Adapit to adopt a new perspective for implementation of MDE as Service based on automated design.

From the point of view of research and practice, these experiences allowed a better understanding of contexts of start-ups and issues for the MDE adoption. To improve our design practices for the next software projects, we looked for prototyping tools used in agile methods. We concluded that designers should work in a high-level of abstraction than MVC-based application models. More importantly, in order to reduce risks of producing wrong features, the method should also consider design tasks for discovery and invention (tasks C and D). Thus, these are the justifications for the introduction of the Evolutionary Prototyping phase in our methodology for MDWE.

Experiences such as the one in SP[1] provided the reasons why we included many interactions with clients in the process. For the worst-case scenarios that present uncertainty in requirements, besides the acceptance tests after the Functional Prototyping phase, we introduced in MockupToME Method two validations with clients that are associated with design tasks. In our position, it is a common mistake to assume that interactions with clients will ever "delay something" in the software development. In fact, the opposite was observed in practice, with books suggesting that frequent validations with clients is good for shortening time-scales in worst-case scenarios (Schwaber, 2004; Shore and Warden, 2008; Sommerville, 2010). Besides, frequent validations reduce the risks of producing wrong features and, in consequence, reducing rework in iteration cycles and also among iterations. Thus, considering worst-case scenarios that we have experienced, our methodology was also conceived to allows client interaction in three phases of prototyping, each one aiming at reducing the risk of producing wrong features for the next.

9.2. Discrepancies in software projects

These software projects present similar MVC layers, as illustrated by the data in Table 2. Table 3 shows some statistical data about each software project. SP[1] is a little bit more complex than SP[2] due to the development of some more complex use cases, such as `Generate billet`, not supported by source code generators. Besides, Table 2 suggests that SP[1] is more complex due to the following reasons: (1) it is larger and includes the support for CRM besides functionalities for ERP that contextualizes the type of system in SP[2]; (2) SP[1] includes a second DSL for the View layer, the Desktop DSL that is implemented in Java Swing, besides JSP that implements the Web DSL used also in SP[2]; (3) SP[1] owns many Lines-Of-Code (LOC) associated with the Desktop view and less source code for Controller and JSP from the Web view than in SP[2]; (4) SP[1] includes source code for the Remote layer, not included in SP[2]; and, (5) In SP[1], the MVC-based architectural models were designed mostly manually with the help of wizards and in SP[2] these models were automatically generated after the execution of the Evolutionary Prototyping Phase.

Table 2
MVC-based layers used in each software project.

Application layers	SP[1]		SP[2]	
Entities and enumerations	64	13.44%	50	25,25%
Validation on the server side	20	4.20%	35	17,68%
Web controllers	11	2.31%	22	11,11%
Data Access Object (DAO)	16	3.36%	35	17,68%
JSP [web view layer]	190	39.91%	56	28,28%
Java swing [forms/tables]	112	23.53%	0	0%
Java swing [window/dialog]	41	8.61%	0	0%
Remote layer	22	4.62%	0	0%
Lines of code (LOC)	SP[1]		SP[2]	
Entities and enumerations	4947	4.38%	5477	14.95%
Validation on the server side	2919	2.58%	4152	11.33%
Web controllers	6283	5.56%	11652	31.81%
Data Access Object (DAO)	11310	10.02%	5441	14.85%
JSP [web view layer]	17249	15.28%	7266	19.83%
Java swing [forms/tables]	60876	53.92%	0	0%
Java swing [window/dialog]	6524	5.78%	0	0%
Remote layer	2781	2.46%	0	0%

Table 3
Data from application in software projects.

Where automated design helped?	SP[1]	SP[2]
Total weeks to conclude each software project	58	44
Average of weeks that MDWE was used	22	20
Best performance for design	27 h	8 h
Average of time-scales in iterations	$\geq$ 4 weeks	1–2 weeks
Generated source code	64%	82%

Table 3 also shows that MDWE was used in 22 weeks for SP[1], while it was used in 20 weeks for SP[2]. This only means that MDWE is used in less than a half of the time of the overall software projects that consumed a total of 57 weeks in SP[1] and 44 in SP[2]. Although not comparable due to differences in underlying implementation technologies and processes, they have the follow-

ing similarities: (1) they were conducted with MDWE, allowing the generation of many functionalities of type CRUD, and (2) they used the Architectural Prototyping and Functional Prototyping phases, with few differences in source code generators (Basso et al., 2013).

We acknowledge that we cannot compare these two software projects. However, as suggests the data shown in Table 3, these software projects presented different time-scales adopted in each one when compared with the team skills shown in Table 1. In this regard, discrepancies between time-scales are clear. Company A successfully executed SP^2 with iterations planned for one to two weeks, with the best performance for modelling reported as eight hours in the last iterations, i.e., after a learning curve period (Basso et al., 2015). With our first approach for MDWE our best performance for design was 27 h. It demanded in SP^1 an experienced designer and developer, while Company A used only non experienced stakeholders. Based on such information, one could expect that, in SP^2, the MDWE approach would present a worst performance than in SP^1. However, the opposite was observed. Other discrepancy is in regard to the activities performed after the source code generation. Company A did not reported issues associated with round-trip engineering. These are benefits/effects that we have not a clear understanding about the causes, as discussed in the next subsection.

9.3. Lessons learnt

Round-trip engineering is still a challenge in regard to tool support (Mussbacher et al., 2014). However, from our newest experience, we confirm that it is not a big issue, as suggested by Hutchinson et al. (2011). We have learned that a multi-layered architecture associated with the good practice of separating what is generated from manual coding mitigates this overhead. Kelly and Tolvanen (2008) make these recommendations too. Besides, based on an industrial survey, Whittle et al. (2013) agrees in this respect. They have not considered this as an "Achilles heel" for the MDE adoption. Thus, through the assimilation of good practices for the development of manual code, Company A did not consider the round-trip engineering a big issue for the execution of our methodology.

Another important good practice is discussed in the literature by Kelly and Tolvanen (2008): generate 100% of the overall application is difficult, if not impossible; instead, teams should focus on full source code generation, i.e., generate 100% of what is designed. They suggest that lifecycles for model transformations will always present a delimited scope of DSLs and a delimited scope for source code generation. These limitations are not considered as reasons for a non adoption, which means that software factories can benefit from MDE without the generation of 100% of final application (Whittle et al., 2013).

These experiences allowed us to observe some benefits promoted by this methodology as follows.

1. To reduce the risks of producing wrong features between iterations, Schwaber (1995) suggests to acquire feedback about what is being produced in cycles of validation. Thus, short time-scales for iterations are preferred to quickly get feedback from clients, as the ones allowed in our approach.
2. A rich set of CRUD templates to generate diverse GUI structures, besides allowing non-experienced modelers to be included in MDWE-based processes, also allows the design of annotated mockups with action semantics that, for some use case patterns, allowed producing working pieces of software that did not required adjustments in source code.
3. Mockups are helpful to get feedback from clients of requirements in the Evolutionary Design phase and, similarly as Rivero et al. (2014), we also noticed that clients feel more comfortable to opine about requirements when experimenting mockups than visualizng UML diagrams representing MVC-based models.

Because our proposal requires the use of transformation templates to generate and refine mockups, there are some drawbacks as follows.

1. Model transformations can fail, meaning that the proposed methodology is only effective if transformation templates are perfectly working.
2. Client can request CRUD structures, or other use case scenarios, not yet developed as transformation templates.
   (a) This would require a manual design of annotated mockups, implying in the development of the use case without the automated design techniques introduce in the Evolutionary Prototyping phase.
   (b) This could also imply in issues for execution of MDWE, such as those observed in SP^1, instead of benefits observed in SP^2.
3. This methodology is only effective if enough transformation templates are available and if they meet the client needs. Otherwise, it became a manual design approach for MDWE, which we did not considered interesting for start-up contexts.

9.4. Open questions

In a previous work we reported some issues and open questions to implement MDE as Service considering the pragmatical aspect of combination of MDWE and Scrum (Basso et al., 2015), summarized as follows: (1) The literature of the area lacks information on how to introduce MDE in specific contexts; (2) Some authors claim UML-based MDWE approaches are "counter agility", but are they really?; (3) The "good" and "bad" on the combination of MDE and Agile should be associated with a context; (4) There is no requirement for "agile tools"; (5) There is no empirical information in the literature on incompatibilities between MDE and Agile Methods/Principles; and (6) Which are the suitable MDE techniques for dealing with round-trip?

The last open question is discussed in this paper with the techniques that we considered interesting. In the following we complement the aforementioned work with research gaps for technical-level issues. For example, the execution of SP^2 presents some benefits not observed in SP^1 such as short time-scales and the mitigation of reverse engineering. For instance, we concluded that such benefits are associated with our methodology for automated design and tool support when executed integrally. However, the reasons for such benefits are not totally clear, thus raising the following open questions:

Is full source code generation the unique reason for mitigation of issues associated with reverse round-trip engineering in SP^2? In our tool support, full source code generation allow to perform changes from model to code following an iterative and incremental process. This is possible only for specific types of use case patterns for web information systems: CRUD, List, Filter, and Report. Since the execution of SP^2 we concluded that, for this type of functionalities, changes performed along iterations would not require the execution of manual round-trip engineering, since they are changed in models and re-generated. Besides, Company A did not considered round-trip engineering as an issue. Currently, we are asking ourselves whether the full source code generation is the unique reason why round-trip engineering is not an issue.

Is our approach good for improve the quality, modularization, and maintenance of source code? Related works present such benefits as promoted by their approaches (Martínez et al., 2011; 2013; Brambilla and Fraternali, 2014; Rivero et al., 2014). Analysing the data shown in Table 2, we conclude that the introduction of tool support in Evolutionary Prototyping phase, adopted

in SP[2], may have influenced the generation of more elements in the MVC layers than in SP[1], thus resulting in more modular and maintainable source code. Paradoxically, this would mean that the automated design techniques, introduced on the top of our method, could lead inexperienced stakeholder to produce an application with more quality than the produced by the two first authors of this article, which are "experts". This question needs further investigation.

Is the multi-layered architecture good to mitigate round-trip engineering? In our approach, developers commit changes quickly from models to the implementation without overriding the manual work made in previous iterations. Allier et al. (2015) state that a design directed to the MVC-based architecture helps on the modularization and organization of the source code. The regeneration of source code includes as input model elements designed conforms to MVC-based application layers. Based on principles of modularity, we recommended that manual coding must be allocated in isolated modules from the generated source code. In this sense, this recommendation associated with a multi-layered design may be responsible for the non existence of big issues for round-trip in Company A. However, although not prepared with good practices discussed above, with the same multi-layered architecture we observed round-trip issues in SP[1]. Thus, this is a paradox that must be investigated.

Is our approach good for requirement discovery and validation? One of the reasons for the development of a new approach for MDWE was our incapacity to perform validations of models in short time-scales in SP[1]. Our clients feel comfortable to opine about requirements represented in paper prototypes, but they have many difficulties to understand and opine about the UML models. Besides, they wanted to click in buttons from real prototypes before provide a feedback of "100% sure" about validity of paper prototypes. We could not do this in SP[1]. The Evolutionary Prototyping Phase was introduced to bridge the requirement engineering and the representation of models associated with MVC layers. In Adapit we observed this as a benefit for the requirement discovery and validation promoted by MockupToME Method. However, Company A reported that they have not experienced a case where requirements changed radically, as occurred within SP[1]. We consider observations made internally in Adapit few to answer the aforementioned question, mainly because there is a tendency to consider this important. Thus, an open question is whether and where tasks associated with discover and invention (C–E) help designers in this transition from the Requirement Engineering Phase to the Evolutionary Prototyping Phase in other software projects.

When developers should not automate acceptance test cases? This question is relevant because we consider that in worst-case scenarios the automation of acceptance test cases may add overhead to the iterations. Likewise, due to frequent changes on some requirements from the worst-case scenarios observed in SP[1], the development of automated acceptance test cases may be non effective. Anyway, assuming that requirements changed, that models must be changed, that the source code must be regenerated, acceptance tests need to be re-executed in a new cycle of acceptance. Our doubt is whether developers should automate the acceptance test cases for this cases. The automation would imply also on the redevelopment of the algorithm for behavior and, as consequence, adding overhead for the execution of iterations in short time-scales. Thus, we are investigating whether these tests can also be generated.

10. Limitations

Limited to some use case patterns. This methodology and tool support are limited for use case patterns of type CRUD, List, Filter and Report. Examples of what we have not yet considered in the automated design includes: top-level layouts for web sites, features from HCI (rich menus, navigation, flows, responsive design), integration with web services, enterprise application integration, and others. Although our work is limited in this regard, several DSLs have been proposed in the literature to represent such abstractions. Thus, they may be included in this methodology conform requests.

No silver bullet. Although MDE is not new, i.e., an MDWE approach dates 2000 (Brambilla et al., 2008), putting it into practice remains a challenge. Mussbacher et al. (2014) have pointed out issues that would be fixed only in the next thirty years from 2014. MDE can work on certain conditions and contexts (Martínez et al., 2011; 2013), such as for the contexts of the reported software projects. However, any MDE approach presents several "Achilles heel" that should be explored in research and practice (Torchiano et al., 2013; Agner et al., 2013; Whittle et al., 2013; Mussbacher et al., 2014). For example, this work is limited for some use case patterns supported by the presented automated design techniques, which means that it is ineffective for other types of use cases. Therefore, this work should never be considered as a silver bullet for software development, needing investigation of feasibility for each context.

11. Related work

We present the related works with the methodology and tool support, considering three phases for prototyping: (1) **Evolutionary Prototyping**, which is classified as an approach for exploratory design; (2) **Architectural Prototyping**, which is classified as a modelling phase dedicated to represent models with more details and in conformity with layers of the MVC; and, (3) **Functional Prototyping**, which is characterized by the generation of full source code for all the layers of the adopted underlying architecture.

The evolutionary prototyping is dedicated to the design of mockups. Balsamic Mockups Company (2015) provides a software tool to represent sketches, without the support for annotations that embed business logic. Blankenhorn (2004); Vanderdonckt (2005) and Kavaldjian (2007) provide similar tools to support the design of mockups with UML Profiles, also without embedding the business logic into GUI components. WebML (Brambilla et al., 2008) and its commercial implementation named WebRatio (Brambilla and Fraternali, 2014) also presents contributions for this phase, allowing the transformation from BPMN flows representing the business model of the application into GUI mockups represented with the WebML. Other DSLs are closer to the MockupToME DSL, such as those provided by Rivero et al. (2014) and Forward et al. (2012), which use annotations in mockups to represent the semantics for business logic. In this sense, Stary (2000) suggests that transformations started from mockup are the key to improve client feedback in preliminary phases of a software process, since they verify acceptance of a given requirement using paper prototypes. Our differential is the introduction of techniques for automated design that speed-up the design of annotated mockups.

To visualise and modify intermediate specifications between mockups and executable prototypes for GUI, Molina et al. (2012) propose an interesting tool namely CIAT-GUI that allows to test information system models in different abstraction layers of application. CIAT-GUI can also be classified as implementing these three phases of prototyping. The differences are that their approach uses a unique DSL while ours use many (e.g., MockupToME DSL, Web DSL, Desktop DSL, Mobile DSL). Although we have not yet tested other DSLs in our tool support than those discussed in this paper, we hope that this feature will enable us to explore/include other possibilities for DSLs and design tools in iso-

lated phases of prototyping. On the other hand, the use of a unique DSL simplifies the execution of the three phases of prototyping, connecting elements based on the same metamodel.

Our methodology also includes resources for model transformation to help designers in the transition from the evolutionary to the architectural prototyping phase. Rivero et al. (2014) present a similar proposal to ours, since that annotated mockups are used as input to generate other application layers. The differences are: (1) we applied the generation of a mockup using start transformation templates, while these authors suggest to manually design mockups; (2) we included a richer support for the execution of automated design techniques; and, (3) we used mockups in the evolutionary phase to explore different possibilities of implementation, while the authors considered a static structure for mockups without options for selection.

Related with the architectural prototyping phase, some works proposes DSLs for the design of web information systems based on the MVC. Souza et al. (2007) presents an approach for MDWE using UML interfaces very similar to those used in our methodology to represent the business logic. Nunes and Schwabe (2006) propose the HyperDE, an environment to produce web information systems by specifying models and transforming them into functional prototypes, starting by a domain model. Similarly, Vara and Marcos (2012) propose a framework composed of a set of model transformations that allows to develop information systems through DSLs. An experience report with the WebRatio (Brambilla and Fraternali, 2014) also presents positive results associated with the source code generation based on architectural models manually specified: a small difference is that WebRatio uses as input a conceptual model representing the data-model for a database while we use a class diagram. Thus, as a small contribution to the practice, our methodology includes wizards that help the designer on the representation of details for MVC-based models.

All these works allows the generation of prototypes. However, only those that are classified as part of the architectural prototyping can also generate fully implemented prototypes.

Yulkeidi, Martinez, Rivero and Brambilla concluded that, in a comparison of MDWE with manual coding, a model-based process improves the productivity and software quality through modularization and maintenance of source code (Martínez et al., 2011; 2013; Rivero et al., 2014; Brambilla and Fraternali, 2014). We have not yet reached these benefits through our analytical studies, more related with the execution of approaches for MDE as Service than specificities of results from MDWE.

Finally, other type of proposal aims at starting prototyping with the specification of many web information systems details with textual DSLs. It is the case for Forward et al. (2012), whose approach is similar to modern frameworks to develop web applications such as Ruby on Rails. These frameworks are used on development phase, not in the evolutionary prototyping. Our methodology is different from theirs since it implements three phases of prototyping based on MDWE, while Forward et al. (2012) used a more direct approach for prototyping. To the best of our knowledge, there is no experimental evidences that suggests that the use of textual DSLs in preliminary software phases is a better solution to perform a requirement analysis than using architectural designs. Thus, this is also an open question that should be investigated in empirical studies.

Rossi (2013) discusses on existing web DSLs, highlighting the importance of a new standard proposed by OMG to design web applications: the Interaction Flow Modeling Language (IFML). IFML standardises several of the representations included by the aforementioned DSLs. This language, as well as WebML and WebRatio, are complementary to MockupToME and overlaps some representations used in the UML Profiles from the Architectural Prototyping phase. A future work will explore this complementarity, presenting our profiles with appropriate comparisons with the state-of-art in MDWE.

Our contribution complements the literature of the area with an integrated approach by methodology and tool support for MDWE. In addition, the reported experiences suggest that the implemented automated design techniques can promote the introduction of MDWE in contexts that present issues for adoption. Thus, we present improvements in practices and tools with fully assisted design tasks for web information systems, which is only partially explored by related works.

12. Conclusions and future work

This paper presents a new MDWE methodology to automate the design of multi-layered web information systems called MockupToME Method. Along the development of some web information systems, we noticed that, for the worst-case scenarios on the requirement engineering (i.e., in start-up contexts), paper prototypes themselves do not ensure the validity about requirements along software process iterations. These specifications change along the iterations, which makes difficult the execution of a MDWE approach. To deal with these chaotic scenarios, we concluded that short duration iterations should be adopted. However, the manual design of MVC-based application models hampers the execution of short time-scales. Thus, we proposed the automation of design tasks.

The MockupToME Method suggests the execution of design tasks and client evaluations about the designed models in three phases: Evolutionary Prototyping, Architectural Prototyping and Functional Prototyping. This execution includes the following features for rapid application prototyping that we consider as benefits for the state-of-practice in MDWE: (1) designers specify annotated mockups with semantics for actions in the Evolutionary Prototyping with the assistance of automated design techniques, supported by model transformations and refinements allowed in a mockup drawing tool; (2) the adoption of concepts such as Master/Detail, DDD, Multi-view, and other, allows the development of different templates for construction and refinement of models represented in different abstractions levels, thus allowing the use by non experienced designer; (3) these techniques are limited for use case patterns of type CRUD, List, Filter and Report; (4) for the worst-case scenarios regarding requirements uncertainty, client and designer interact in tasks for discovery and invention, e.g., while constructing and updating a model specification, they are allowed to visualize different implementation options for a use case to decide which of them fits best to the needs of the iteration; (5) the discovery and invention is considered as important for clients to reach more necessities, which is good in MDWE, allowing designers to quickly change designed models before execute the Architectural Prototyping and Functional Prototyping phases; and, (6) these features, added to other elements such as source code generation and practices discussed in this paper, allowed for the execution of iterations lasting one week.

We summarized two industrial experiences in the development of web information system using our proposal in piece and integral. In the first experience, which adopted mostly a manual design approach, we observed many issues for execution of the software project including the long time invested in manual representation of models, issues in source code generation and bad practices for manual coding. Moreover, changes in requirements, motivated by misunderstanding or simply because the client decided to adopt other features for innovation, consumed too much time from the overall software project. We concluded that MDWE issues associated with these changes such as round-trip engineering and rework in two levels of abstractions (models and code) could

be related with the execution of iterations, planned and executed with more than one month. However, due to the incapacity in our tool support and practices adopted in 2007, we could not perform shorter iterations than a month.

The need for execution of shorter time-scales is the main reason for the development of the proposed methodology. Likewise, the most recent experience presented promising results promoted by our proposal, such as the possibility of execution of iterations lasting one to two weeks. The reasons for such a benefit are not totally clear for us. However, automated design techniques are clearly related. Other features that can be related include our recommendations for manual coding, client validations executed in three phases of prototyping, full source code generation, modularity promoted by a multi-layered MVC structure, the introduction of a phase for discovery and invention and the context of the developed system. Thus, we also addressed these features as open questions relevant for the theory and practice of MDWE.

To have a clear notion about the reasons why MockupToME Method is capable for execution of short duration sprints, we will conduct new works as follows:

- Conduct a study in retrospective considering projects executed with different approaches for MDWE. Accordingly, we will mine repositories from five software projects that used partially and integrally the tasks and tools associated with the MockupToME Method. We believe that, by mining these repositories, we can find answers for our open questions.
- Execute a second study for evaluation of the quality attribute "productivity" in agile teams. In a previous study that aimed at compare the productivity of two agile teams (Basso et al., 2014d), one adopting our methodology and tool support and the other developing the software without MDWE, we could not reach strong conclusions. This is because the study presented confounding factors such as differences on the underlying implementation framework and lacks of quantitative data. Thus, a future work will apply this methodology in another agile context to measure this quality attribute.
- Highlight our technical contributions, presenting details of associated scripts for model transformations and metamodels. So far, our contributions discusses only aspects associated with the management and reuse of model transformation components (Basso et al., 2013). Our long-term goal for future works is to discuss particularities from our metamodels and tool support, thus presenting some contributions for the state-of-art in MDWE.

Acknowledgments

The research work on which we report in this paper is supported by CNPQ and Capes-Brazil (first three authors), and by the internal Research Programme 2012/13 at UNIJUI University (fourth and fifth authors).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2016.04.060.

References

Agner, L.T.W., Soares, I.W., Stadzisz, P.C., Simao, J.M., 2013. A brazilian survey on uml and model-driven practices for embedded software development. J. Syst. Softw. 86 (4).

Allier, S., Barais, O., Baudry, B., Bourcier, J., Daubert, E., Fleurey, F., Monperrus, M., Hui, S., Tricoire, M., 2015. Multitier diversification in web-based software applications. Softw., IEEE 32 (1), 83–90.

Ambler, S.W., 2015. A Roadmap for Agile MDA. Technical Report. Agile Modeling.Available at: http://www.agilemodeling.com/essays/agileMDA.htm

Balsamic Mockups Company, 2015. Balsamiq Mockups Company URL: https://balsamiq.com/products/mockups/.

Basso, F.P., Becker, L.B., Oliveira, T.C., 2007. Uma solução para reuso e manutenção de transformadores de modelos usando a abordagem fomda. In: Simpósio Brasileiro de Engenharia de Software. Anais do 21o Simpósio Brasileiro de Engenharia de Software, pp. 130–146.

Basso, F.P., Oliveira, T.C., 2007. WorkCASE Toolkit: Uma Ferramenta de Suporte para Agile Model Driven Architecture. Technical Report. Adapit Soluções em TI.

Basso, F.P., Oliveira, T.C., Farias, K., 2014a. Extending junit 4 with java annotations and reflection to test variant model transformation assets. In: Proceedings of the 29th Symposium On Applied Computing, pp. 1601–1608.

Basso, F.P., Pillat, R.M., Frantz, R.Z., Rooz-Frantz, F., 2014b. Assisted tasks to generate pre-prototypes for web information systems. In: Proceedings of the 16th International Conference on Enterprise Information Systems, pp. 14–25.

Basso, F.P., Pillat, R.M., Oliveira, T.C., Becker, L.B., 2013. Supporting large scale model transformation reuse. In: Proceedings of the12th International Conference on Generative Programming: Concepts & Experiences, pp. 169–178.

Basso, F.P., Pillat, R.M., Oliveira, T.C., Fabro, M.D.D., 2014c. Generative adaptation of model transformation assets: experiences, lessons and drawbacks. In: Proceedings of the 29th Symposium On Applied Computing, pp. 1027–1034.

Basso, F.P., Pillat, R.M., Roos-Frantz, F., Frantz, R.Z., 2015. Combining mde and scrum on the rapid prototyping of web information systems. Int. J. Web Eng. Technol. 10 (3), 214–244.

Basso, F.P., Pillat, R.M., Rooz-Frantz, F., Frantz, R.Z., 2014d. Study on combining model-driven engineering and scrum to produce web information systems. In: Proceedings of the 16th International Conference on Enterprise Information Systems, pp. 137–144.

Batory, D., Latimer, E., Azanza, M., 2013. Teaching model driven engineering from a relational database perspective. In: Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems, pp. 121–137.

Blankenhorn, K., 2004. A UML Profile for GUI Layout. University of Applied Sciences Furtwangen. Department of Digital Media Master's thesis. URL: http://www.bitfolge.de/pubs/thesis/.

Booch, G., Rumbaugh, J., Jacobson, I., 2005. The Unified Modeling Language User Guide (2nd Edition). Addison-Wesley.

Bosch, J., 2013. Achieving simplicity with the three-layer product model. IEEE Comput. 46 (11), 34–39.

Brambilla, M., Fraternali, P., 2014. Large-scale model-driven engineering of web user interaction: the webml and webratio experience. Sci. Comput. Program. 89, Part B (0), 71–87.

Brambilla, M., Fraternali, P., Tisi, M., 2008. A metamodel transformation framework for the migration of webml models to mda. In: CEUR-WS Proceedings of the 4th International Workshop on Model-Driven Web Engineering (MDWE 2008). volume 389, Tolouse, France., pp. 91–105.

Burke, B., Monson-Haefel, R., 2006. Enterprise JavaBeans 3.0: Developing Enterprise Java Components. O'Reilly.

Davis, F., Venkatesh, V., 2004. Toward preprototype user acceptance testing of new information systems: implications for software project management. IEEE Trans. Eng. Manag. 51 (1), 31–46.

EDOC, 2014. UML Profile For Enterprise Distributed Object Computing (EDOC) URL: http://www.omg.org/spec/EDOC/.

Evans, E., 2004. Domain-DrivenDesign: Tackling Complexity in the Heart of Software. Addison Wesley.

Forward, A., Badreddin, O., Lethbridge, T., Solano, J., 2012. Model-driven rapid prototyping with umple. Softw.: Pract. Exp. 42 (7), 781–797.

France, R.B., Bieman, J.M., 2001. Multi-view software evolution: a UML-based framework for evolving object-oriented software. In: ICSM, pp. 386–395.

Gärtner, M., 2012. ATDD by Example: A Practical Guide to Acceptance Test-Driven Development. Addison-Wesley Signature Series (Beck) 1st Edition.

Giardino, C., Unterkalmsteiner, M., Paternoster, N., Gorschek, T., Abrahamsson, P., 2014. What do we know about software development in startups? Softw. IEEE 31 (5), 28–32.

Han, H., Liu, B., 2010. Problems, solutions and new opportunities: using pagelet-based templates in development of flexible and extensible web applications. In: Proceedings of the 12th iiWAS'10, pp. 679–682.

Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S., 2011. Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 471–480.

Kavaldjian, S., 2007. A model-driven approach to generating user interfaces. In: Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, pp. 603–606.

Kelly, S., Tolvanen, J.-P., 2008. Domain Specific Modeling: Enabling Full Code Generation. IEEE Computer Society - John Wiley & Sons.

Kent, S., 2002. Model driven engineering. In: Integrated Formal Methods, pp. 286–298.

Kulkarni, V., Barat, S., Ramteerthkar, U., 2011. Early experience with agile methodology in a model-driven approach. In: Proceedings of the 14th International Conference on Model-Driven Engineering Languages and Systems, pp. 578–590.

Landre, E., Wesenberg, H., Olmheim, J., 2007. Agile enterprise software development using domain-driven design and test first. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, pp. 983–993.

Linington, P.F., 2005. Automating support for e-business contracts. Int. J. Cooperative Inf. Syst. 14 (2–3), 77–98.

Martínez, Y., Cachero, C., Matera, M., Abrahao, S., Luján, S., 2011. Impact of mde ap-

proaches on the maintainability of web applications: an experimental evaluation. In: Conceptual Modeling - ER 2011. In: Lecture Notes in Computer Science, Vol. 6998. Springer Berlin Heidelberg, pp. 233–246.
Martínez, Y., Cachero, C., Meliá, S., 2013. MDD vs. traditional software development: a practitioner's subjective perspective. Inf. Softw. Technol. 55 (2), 189–200.Special Section: Component-Based Software Engineering (CBSE), 2011
Moe, N.B., Dingsoyr, T., Dyba, T., 2010. A teamwork model for understanding an agile team: a case study of a scrum project. Inf. Softw. Technol. 52 (5), 480–491.
Molina, A.I., Giraldo, W.J., Gallardo, J., Redondo, M.A., Ortega, M., García, G., 2012. Ciat-gui: a mde-compliant environment for developing graphical user interfaces of information systems. Adv. Eng. Softw. 52, 10–29.
Molina, P.J., Meliá, S., Pastor, O., 2002. Just-ui: A user interface specification model. In: Computer-Aided Design of User Interfaces III, pp. 63–74.
Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B.H., Collet, P., Combemale, B., France, R.B., Heldal, R., Hill, J., Kienzle, J., Schöttle, M., Steimann, F., Stikkolorum, D., Whittle, J., 2014. The relevance of model-driven engineering thirty years from now. In: Model-Driven Engineering Languages and Systems, pp. 183–200.
Nunes, D.A., Schwabe, D., 2006. Rapid prototyping of web applications combining domain specific languages and model driven design. In: Proceedings of the 6th International Conference on Web Engineering, pp. 153–160.
Parnas, D., 1994. Software aging. In: Proceedings of the 16th International Conference on Software Engineering, pp. 279–287.
Pillat, R.M., Oliveira, T.C., Alencar, P.S., Cowan, D.D., 2015. BPMNt: a BPMN extension for specifying software process tailoring. Inf. Softw. Technol. 57 (0), 95–115.
Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., Astesiano, E., 2010. On the effort of augmenting use cases with screen mockups: results from a preliminary empirical study. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 40:1–40:4.
Rivero, J.M., Grigera, J., Rossi, G., Luna, E.R., Montero, F., Gaedke, M., 2014. Mockup–driven development: providing agile support for model-driven web engineering. Inf. Softw. Technol. 56 (6), 670–687.
Rossi, G., 2013. Web modeling languages strike back. Internet Comput., IEEE 17 (4), 4–6.
Schmidt, D.C., 2006. Guest editor's introduction: model-driven engineering. IEEE Comput. 39 (2), 25–31.
Schwaber, K., 1995. Scrum development process. In: Workshop on Business Object Design and Implementation, OOPSLA'95, pp. 1–23. URL: http://agilix.nl/resources/scrum_OOPSLA_95.pdf
Schwaber, K., 2004. Agile Project Management with Scrum (Microsoft Professional). Microsoft Press.
Shore, J., Warden, S., 2008. The Art of Agile Development. O'Reilly.
Sommerville, I., 2010. Software Engineering (9th Edition). Addison-Wesley.
Souza, V.E.S., Falbo, R.D.A., Guizzardi, G., 2007. A UML profile for modeling framework-based web information systems. In: Proceedings of the 12th International Workshop on Exploring Modelling Methods in Systems Analysis and Design EMMSAD '2007, pp. 153–162.
Stary, C., 2000. Contextual prototyping of user interfaces. In: Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques, pp. 388–395.
Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., Reggio, G., 2013. Relevance, benefits, and problems of software modelling and model driven techniques-a survey in the italian industry. J. Syst. Softw. 86 (8), 2110–2126.
Vanderdonckt, J., 2005. A MDA-compliant environment for developing user interfaces of information systems. In: Proceedings of the 17th International Conference on Advanced Information Systems Engineering, pp. 16–31.
Vara, J.M., Marcos, E., 2012. A framework for model-driven development of information systems: technical decisions and lessons learned. J. Syst. Softw. 85 (10), 2368–2384.
Voelter, M., 2009. Best practices for dsls and model-driven development. J. Object Technol. 8 (6), 79–102.
Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R., 2013. Industrial adoption of model-driven engineering: are the tools really the problem? In: Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems, pp. 1–17.
Zhang, Y., Patel, S., 2011. Agile model-driven development in practice. Softw. IEEE 28 (2), 84–91.

Fábio Paulo Basso is currently a PhD student at Federal University of Rio de Janeiro (COPPE/UFRJ), Brazil. His effort is on the technical feasibility of Model-Driven Engineering applied as Service in startup contexts, including topics such as Domain Specific Languages and adaptive support for Model Transformation Chains.

Raquel Mainardi Pillat is currently a PhD student in Software Engineering at Federal University of Rio de Janeiro (COPPE/UFRJ), Brazil. Her research interests include software processes, Model-Driven Engineering and tailoring of models represented with the BPMN.

Toacy Oliveira is currently an Assistant Professor at Federal University of Rio de Janeiro (COPPE/UFRJ), Brazil. He is also Adjunct Professor with the David R. Cheriton School of Computer Science at the University of Waterloo, Canada. He received his education at Pontifical Catholic University of Rio de Janeiro, Brazil (Electrical Engineering -1992, MSc-1997, PhD - 2001) and spent 3 years at University of Waterloo as a posdoc. His current research interests are under the software engineering umbrella, including software processes and software reuse. Toacy focuses on the use notations, models, processes and tools to improve the way software systems are developed. He has published over 50 refereed publications, and has been a member of program committees of numerous highly-regarded conferences and workshops. He has also been a leading investigator in national projects supported by CAPES and CNPq.

Fabricia Roos-Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the UNIJUÍ University, Brazil. She received her PhD in Software Engineering from the University of Seville, Spain. Her current research interests include software product lines and search-based software engineering.

Rafael Z. Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the Unijui University, Brazil, and leads the Applied Computing Research Group since 2013. He was awarded a PhD degree in Software Engineering by the University of Seville, Spain. His current research interests focus on the integration of enterprise applications and search-based software engineering.