Softw Syst Model (2019) 18:663–690
https://doi.org/10.1007/s10270-017-0586-9

REGULAR PAPER

DataMock: An Agile Approach for Building Data Models from User Interface Mockups

José Matías Rivero[1,2] · Julián Grigera[1] · Damiano Distante[3] · Francisco Montero[4] · Gustavo Rossi[1,2]

Received: 13 December 2015 / Revised: 17 January 2017 / Accepted: 20 January 2017 / Published online: 14 February 2017

Abstract In modern software development, much time is devoted and much attention is paid to the activity of data modeling and the translation of data models into databases. This has motivated the proposal of different approaches and tools to support this activity, such as semiautomatic approaches that generate data models from requirements artifacts using text analysis and sets of heuristics, among other techniques. However, these approaches still suffer from important limitations, including the lack of support for requirements traceability, the poor support for detecting and solving conflicts in domain-specific requirements, and the considerable effort required for manually checking the generated models. This paper introduces DataMock, an Agile approach that enables the iterative building of data models from requirements specifications, while supporting traceability and allowing inconsistencies detection in data requirements and specifications. The paper also describes how the approach effectively allows improving traceability and reducing errors and effort to build data models in comparison with traditional, state-of-the-art, data modeling approaches.

Keywords Data modeling · Agile methods · Mockups · Annotations · Requirements engineering · Requirements traceability · Model-driven development

Communicated by Dr. Benoit Baudry.

✉ Damiano Distante
damiano.distante@unitelma.it

José Matías Rivero
mrivero@lifia.info.unlp.edu.ar

Julián Grigera
julian.grigera@lifia.info.unlp.edu.ar

Francisco Montero
fmontero@dsi.uclm.es

Gustavo Rossi
gustavo@lifia.info.unlp.edu.ar

1 LIFIA, Facultad de Informática, Universidad de La Plata, La Plata, Argentina

2 Conicet, La Plata, Argentina

3 Unitelma Sapienza University, Rome, Italy

4 Universidad de Castilla-La Mancha, Albacete, Spain

1 Introduction

Data modeling, i.e., the process of defining and analyzing data requirements and creating data models that satisfy them, is an essential and unavoidable task in the development of software systems [1]. This activity is critical when developing all kind of interactive software, e.g., Web and mobile applications, which have sophisticated User Interface behaviors and where the presented content is retrieved from databases or services reflecting the underlying domain model. Even in applications not involving very complex or huge amounts of data, the modeling process implies the understanding of aspects related to the representation of data in an appropriate way (e.g., through an UML class model).

Building correct data/class models is not easy, as it involves complex and critical activities such as: (i) collecting correct and complete data requirements from stakeholders, (ii) understanding the details of the application's underlying domain, (iii) properly analyzing requirements to define data models, and (iv) mastering the formalism chosen to define data models. To make things worse, stakeholders usually express requirements using a business-related jargon, often unknown to developers or analysts beforehand.

While stakeholders, particularly end-users, closely participate in the requirements gathering stage, analysts usually end up with semistructured text-based requirements specifications, like UML Use Cases or User Stories. Such specifications are then manually or semiautomatically transformed into data models represented using one of the most adopted formalisms for data modeling, such as Entity-Relationship diagrams, Relational models, or UML Class Diagrams [1,2].

Manually building data models from requirements specifications does not enable *requirements traceability*. This means that the relationships between the derived data model and the requirements from which they originated will not be kept over time, as the requirements or the generated models and software artifacts evolve [3,4]. A traceability loss can lead to incorrect or inconsistent data models, with missing or unnecessary elements in them.

To help developers through the data modeling process, different techniques and supporting tools have been proposed in the literature for *generating* data models from requirements specification provided in the form of free or structured text and expressed in natural language [5–7]. However, these techniques have some limitations like, for instance, difficulty in preserving traceability between those specification, limitations due to the lack of rigid structure in free text or to low understandability for end-users of formal structures. The lack of support for requirements traceability and conflicts detection usually requires a high effort for manually checking the generated models [7–9].

To overcome the aforementioned limitations of state-of-the-art data modeling techniques, we propose *DataMock*, an Agile approach for generating data models from requirements specification that supports requirements traceability and data model inconsistencies detection and resolution.

With *DataMock*, data models are derived from data requirements with an iterative process that transforms data requirements specified in the form of User Interface *mockups* (*UI mockups*) and mockup *annotations* into data models represented in the form of UML Class Diagrams [10]. Tool support is provided for the different steps of the process and data models are generated in a semiautomatic way thanks to the use of model-driven engineering techniques.

Mockups are low-fidelity graphical prototypes commonly used, particularly in Agile software development methodologies, as a "quick and dirty" way of gathering and specifying requirements for a software to be developed [11–13]. They can be used to specify different types of requirements, including those related to the appearance (User Interface layout, presentation and interaction widgets), the behavior (user interaction, workflows, business rules, etc.), and the data (domain model, content navigation structure, etc.) of a software product from an end-user perspective [14]. One of the features that motivate the adoption of mockups in the software development process is the clear understandability of their concepts, for both developers and end-users, which makes them a natural *shared language* and a mean to overcome the *business jargon* problem [15]. This is made possible by the use of visual metaphors (e.g., windows, buttons, links, text boxes, popups) that are familiar to end-users, and at the same time clearly understandable by developers, along with their technical implications. In addition, recently conducted empirical studies support the presumption that mockups improve the software development process in general [11,16], especially when used as a foundation for application modeling [17]. Moreover, mockups are the most used requirements artifact in Agile methodologies [11], which are in turn the most adopted in the industry, according to recent surveys.[1]

DataMock represents a novel approach to data modeling in which the data model is built iteratively by deriving atomic data elements from mockups, thus providing an explicit traceability and validation against them. In addition, this iterative, mockup-binding modeling approach also allows detecting errors in data requirements specifications. The approach described in this paper is not a fully fledged Model-Driven Engineering approach, but it introduces a Domain-Specific Language based on formal annotations and a set of UI widgets to assist developers in the task of building data models in a faster and less error-prone way. However, since it allows to generate data models in industry well-known interchange formats, the approach can be integrated to other Model-Driven approaches.

The rest of the paper is organized as follows. Section 2 provides the background and discusses some works related to ours. Section 3 presents our approach for data modeling in depth, providing details about its theoretical, procedural, and technical aspects. Section 4 describes a controlled experiment conducted to prove the applicability and feasibility of the approach and to assess its added value. Finally, Sect. 5 concludes the paper and introduces future works we are pursuing.

2 Background and Related Work

Data modeling is a fundamental part of the process to develop an information system [1]. Classic, full code-centric methodologies generally rely on data models. Scaffolding (or so-called Rapid Application Development) approaches that allow to automatically generate part of the application code, such as Ruby-On-Rails[3] and Grails[4], also require

[1] The 9th Annual State of Agile™ Survey. http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf.

data specifications as an input (e.g., through commands or annotations). This is also a requirement in more model-based approaches like the ones presented in [18,19]. Even methodologies that specify applications at a high abstraction level like Model-Driven Web Engineering (MDWE) methodologies—such as Interaction Flow Modeling Language (IFML) [20] and UML-Based Web Engineering (UWE) [21], heavily rely on data models as a foundational part of their modeling processes. In particular, MDWE methodologies, although effective, require creating data models at the beginning, in order to specify other artifacts that rely on them, such as the navigation and presentation models of the Web application. Finally, all these approaches (ranging from code-centric to full model-driven solutions) generally require the manual translation of free or semistructured textual requirements into code or data model representations and do not provide ways of maintaining traceability between requirements and data model concepts. In addition, they do not provide automatic ways to detect data requirements inconsistencies. Software requirements elicitation and modeling are key activities in software development, but requirements-related errors are yet a challenge [22].

Generating conceptual models from structured form-like User Interface specifications has been already proposed by Ramdoyal et al. [23]. In this work, simple interface widgets allow users to express concepts without considering potentially irrelevant User Interface aspects like detailed layout and presentation or interaction details. The set of possible widgets allowed in this approach are only the most commonly used in forms (like input boxes, buttons, tables). Each widget is mapped to an Entity-Relationship (E-R) concept, and heuristics are used along the drawing process to help generating detailed data models from them. However, since this approach can only be used on form-centered User Interfaces, it cannot be applied in the context of modern User Interfaces (e.g., those that implement Rich Internet Applications behavior, responsive design, etc.). The ICONIX process [24] proposes to start with graphical User Interface (GUI) prototypes as a first requirements artifact. These prototypes are used to gather behavioral requirements and, at the same time, to build a first version of a domain model which contains unambiguous concepts that should be clear for both developers and stakeholders. However, no specific tooling or guidance is provided for generating such models from the GUI prototypes.

Following the same ideas of our approach, in [25] data model specifications are integrated with an existing and well-known requirements artifact: UML Use Cases [26]. In this work, formal flows of information exchanged between the actors and the system in each use case are used to specify data concepts. Since stakeholders participate in the specification of the use cases, in the context of this approach they transitively participate in the construction of the data model. However, the approach requires fulfilling a specific syntax that can be very difficult to understand to end-users or other stakeholders.

Data models generation from requirements specification using Natural Language Processing is also an extensively studied field. In [6] a stepwise conceptual model generation from natural language requirements sentences is proposed. In this work, some of the most important approaches in that field are also referenced. A similar approach that uses the Semantics of Business Vocabulary and Business Rules (SBVR) OMG standard [27] as an intermediate model is commented in [28]. More complex approaches that combine words, sentences, and semantic tagging are also described in [5]. While this kind of approaches provides a semiautomatic way of generating conceptual model elements, the generated models have to be manually refined to obtain their definitive version. Moreover, since they rely on natural language, which usually contains business or even end-user jargon, entities in the final model can contain wrong names or can even be not valuable at all. Requirements traceability, an issue that is considered by our approach, is also a much investigated field. An extensive theoretical, practical, and technological survey can be found in [29].

Data model consistency is also an actively investigated topic [30,31]. According to [31], there exists five types of model consistency: inter-model (consistency between models of different levels of abstraction), intra-model (consistency between models of the same level of abstraction), evolutionary (consistency between different versions of the same model), semantic and syntactic (consistency is validated against semantic and syntactic specifications defined in UML metamodels). However, existing solutions in this field are based on structural model analysis and require domain-based constraints (usually expressed through additional models of a different type) in order to detect domain-related inconsistencies. A detailed analysis and comparison chart of these solutions can be found in [31]. The modeling strategy proposed in our approach belongs to the intra-model set but does not require defining additional and different models for consistency checking. As it will be commented later in the paper, the set of data specifications defined with DataMock has in fact a level of redundancy that allows to check consistency between them.

On the other side, User-Centered Design (UCD) methods provide several improvements in the development process, in particular in the context of Agile methodologies [11]. Among the different human–computer interaction (HCI) techniques used in UCD, User Interface prototypes (mockups) are the most used [11,32,33]. UI prototyping in the software development process is itself a topic that has been studied in detail. In particular, annotating User Interface prototypes has been already described by Constantine et al. [34]. Advantages of using User Interface prototypes in the context of well-known Agile processes have been also reported [12,13,35]. Participatory Design is a design approach of the UCD process where all stakeholders are actively involved in the design processes. Traditional design processes commonly involve clients and consultants; in Participatory Design, end-users are also recognized as stakeholders and are brought into the process as well. Participatory Design and prototyping approaches have proven to be valuable to the development of software [36]. The enrichment strategy through annotations that is further described in this paper can be applied to a variety of requirements artifacts. However, because of the popularity gained by mockups during the last years, the approach makes a foundational use of mockups, thus inheriting all the aforementioned advances already reported in the literature and in industrial surveys.

Conceptual modeling techniques and patterns have been proposed in the literature to help developers and analysts in applying good practices, and well-proven elegant data modeling solutions when building conceptual models [37]. Some of these approaches use type dependencies combined with rules to derive classes and relationships [38]. Conceptual Modeling Patterns undoubtedly help analysts and developers to build better data models applying classic and successful modeling solutions for common and repetitive modeling problems. Thus, they can help to reduce or early detect data requirements inconsistencies in comparison with working with no pattern guidance. However, the applicability of these patterns has to be identified manually by developers or analysts from stakeholders' requirements descriptions and they do not solve the traceability issues mentioned earlier in the paper. Mockup-Driven Development (MockupDD) [17], an approach from which DataMock was defined, already introduced the idea of using and formalizing mockups in order to model and generate software artifacts, working also as a requirements engineering approach [39]. However, MockupDD is a general approach, meant to model a variety of aspects of applications like navigation, object persistence, business rules, detailed queries. While data modeling was also conceived in that approach, it was very limited: Basic aspects like association cardinalities, inheritance, and detailed data types were not included in the language. Thus, data models that can be generated in that approach were very limited. Additionally, no conflict detection and data model generation were provided by the tooling. Data model generation and downloading were not provided either. Finally, the controlled experiment included in that work was general and not focused on the advantages of using mockups for data modeling. This work in which the approach and tooling was focused on data modeling intends to solve all these weak points.

More close to implementation there exist several APIs that allow specifying data modeling concepts directly in the code, using annotation capabilities of modern languages like Java and C#. Two different examples are Java Persistence API (JPA)[2] and those included in the Entity Framework (.NET).[3] While these annotation-based approaches can look similar to the one used in DataMock, there are core differences:

1. They are language based and not technology independent;
2. They are focused on mapping existing object-oriented code to relational databases, generating the DB schema automatically too;
3. They require having an existing model written in an object-oriented language to start defining such relational database-oriented mapping;
4. Underlying artifacts over which these annotations are applied (code) are not understandable by common stakeholders;
5. Annotations, in most cases, are limited to an atomic specification in the sense that they map a concrete OO model element (class, accessor, property, etc.) to a concrete database element. DataMock, instead, allows to specify several data model elements in a single annotation.

In addition to the aforementioned, software designers are faced with challenges due to ever-increasing complexity and market pressures [40]. The ability to efficiently explore design alternatives and to detect design errors as early as possible in the design process is critical to creating high-quality products within short development timeframes.

In DataMock, the data modeling task is accomplished by annotating features required for the application and considered important by its end-users over UI mockups, iteratively. To be consistent with the terms used in our previous work, such annotations will be called *tags* throughout the rest of the paper. The iterative process allows managing the complexity of requirements elicitation activity, while the *tagging* of the requested features directly over concrete requirements artifacts (i.e., mockups) allows preserving requirements traceability [41]. As we will describe later in the paper, this iterative modeling approach also allows automatically detecting requirement conflicts and inconsistencies.

The main advantages of our DataMock data modeling approach can be summarized as follows:

1. It provides pre- and post-requirements specification data requirements traceability [4] by explicitly linking User Interface concepts to data specifications.

[2] TopLink JPA Annotation Reference—http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-096251.html.

[3] Code First Data Annotations—https://msdn.microsoft.com/en-us/data/jj591583.aspx.

Table 1 Feature summary of the different data modeling and data model generation approaches commented in this section

Tool/Method	Modeling strategy	Requirements artifacts supported	Traceability features	Requirements conflict detection
Manual modeling with E-R diagrams, Relational models, OO/UML Diagrams, using traditional requirements artifacts (e.g., Use Cases, User Stories, Mockups)	Manual (model built by hand)	Any	Manual	Manual
Natural Language Processing	Derived, by processing textual requirements artifacts	Text	Weak—models usually require adjustments which imply unavoidable traceability losses	Possible
Processing—e.g., Ramdoyal et al.	Derived, by processing structured UI definitions	Structured UI	Manual	Possible
Modeling over other artifacts (UML)—e.g., Kulak et al.	Manual, using UML Class Diagrams formalism	UML Class Diagrams	Strong—semantics are enough to avoid implementing changes in data models manually	Manual
DataMock	Derived, from tagged UI mockups	Mockups/structured Mockups	Strong—semantics are enough to avoid implementing changes in data models manually	Automatic

2. It improves data modeling productivity by using a Domain-Specific Language (DSL), based on simple and atomic data specifications, to iteratively construct data models.
3. It reduces modeling errors by checking that each modeled concept maps to concrete functional parts of the application, earlier defined through UI mockups.
4. It inherits all well-proven advantages of using mockups in the development process [16,17].

A summary of the characteristics of data modeling approaches commented in this section compared with DataMock is reported in Table 1.

3 The DataMock Approach: From User Interface Mockups to Data Models

In this section, we describe the DataMock approach in detail. First, we introduce its process and general steps. Then, we comment the tags' syntax and application, which is the core of the approach. After introducing such features, we describe how these tags are processed in order to, for instance, generate Class Diagrams and help detecting errors in requirements specifications. Finally, we describe the tool support developed for the approach.

Figure 1 provides an overview of the DataMock approach. In particular, the figure represents the different steps of the data modeling process, the involved actors, the produced data specification and data modeling artifacts, and the provided tool support.

An iteration of the process starts by defining a set of User Stories [42] and one or more mockups associated to them (Steps 1 and 2). A User Story is a textual requirement artifact following the form *As a <role>*, *I want to <goal/desire> so that <benefit>*, being the *<benefit>* part optional. User Stories must be *independent* (i.e., self-contained), *negotiable* (rewritable), *valuable* (i.e., it must deliver value to end-users), *estimable* (i.e., it must have a predictable size), *scalable* (i.e., it must be small sized so that it can be tackled with some estimated effort), and *testable* (i.e., it must provide for being tested after it has been developed).

After all User Stories have been written, developers, with essential help of end-users, apply tags over the mockups associated to every User Story with the data specifications that express the data elements represented on each mockup (Step 3). After each tagging session, developers can generate a new version of the data models (expressed as UML Class Diagrams) of the application being designed in order to check and validate the defined specifications (Step 4). Any refinement required over the data specifications can be done directly over the tagged mockups, since the tagging language (described later in this section) covers the essential modeling aspects.

Tool support is provided for all the steps of the DataMock process as a combination of existing tools and a set of cus-

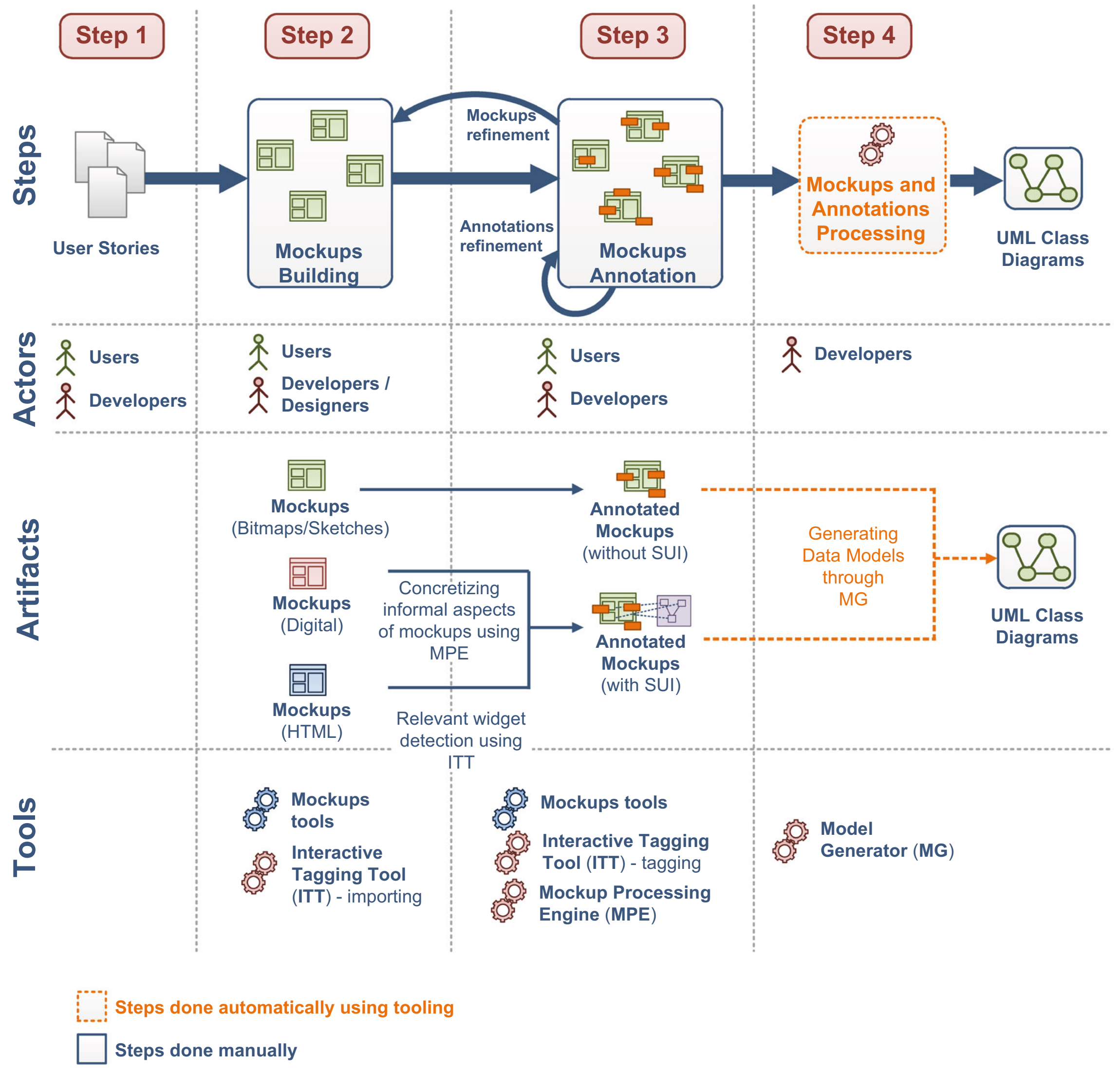

Step 1
Step 2
Step 3
Step 4
Steps
User Stories
Mockups Building
Mockups refinement
Annotations refinement
Mockups Annotation
Mockups and Annotations Processing
UML Class Diagrams
Actors
Users
Developers
Users
Developers / Designers
Users
Developers
Developers
Artifacts
Mockups (Bitmaps/Sketches)
Mockups (Digital)
Mockups (HTML)
Concretizing informal aspects of mockups using MPE
Relevant widget detection using ITT
Annotated Mockups (without SUI)
Annotated Mockups (with SUI)
Generating Data Models through MG
UML Class Diagrams
Tools
Mockups tools
Interactive Tagging Tool (ITT) - importing
Mockups tools
Interactive Tagging Tool (ITT) - tagging
Mockup Processing Engine (MPE)
Model Generator (MG)
Steps done automatically using tooling
Steps done manually

Fig. 1 DataMock iterative data modeling process with actors, tools, and artifacts

tom tools that we developed for the purpose. User Stories (Step 1) can be defined in the form of free text using any text editor. Mockups can be built from User Stories (Step 2) using common digital tools, such as Balsamiq[4] or Pencil[5]—thus creating what we call *tool-based mockups*. Also, HTML mockups built with common WYSIWYG tools or even mockups in the form of digitalized images (what we call *bitmap mockups* in this paper) can be used. Tags can be added to mockups (Step 3) using the tagging features provided by these tools or via our Interactive Tagging Tool (ITT), which allows importing and interactively tagging mockups. If tool-based mockups are used, our Mockup Processing Engine (MPE) tool can be used to concretize some of the informal aspects of mockups (e.g., lack of UI structure) to improve the generation of data models from them. Finally, the processing of mockups to generate data models in the form of UML Class Diagrams (Step 4) can be accomplished using our Model Generator (MG) tool.

3.1 Steps 1 and 2: User Stories Definition and Mockup Building

Since User Stories are mere textual artifacts, they can be defined in any digital or physical way. Once defined, at least one mockup must be associated to every story. Currently, the use of digital tools like Balsamiq or Pencil is a growing trend for building UI mockups. DataMock supports the use of such tools as well as mockups in the form of HTML prototypes, or even digitalized versions of mockups drawn by hand (bitmap mockups). HTML mockups can be reused in DataMock via an interactive tool (described later in the

[4] Balsamiq Mockups. http://balsamiqmockups.com.

[5] Pencil Project. http://pencil.evolus.vn.

paper) that allows filtering and selecting the parts of the HTML source code that are relevant from the data modeling point of view. Bitmap mockups can be reused just importing them in a digital mockuping tool and placing tags using the annotation features already present in such kind of tooling. When using tools like Balsamiq or Pencil with their native widget set or when HTML mockups are manually enhanced filtering the relevant UI components, an initial widget model can be generated from them that results useful in further steps in the approach, as we will describe later. The motivation behind the flexibility for using different kind of mockups is to provide the most extensive support as possible regarding the UI prototyping technology to which both developers and end-users feel comfortable with.

One identified problem when trying to reuse mockups as specification artifacts is their lack in details inherent to the UI structure [43]. If bitmap mockups are used, no structure is defined at all. When using tool-based mockups, their internal structure is usually stored as a set of styled rectangles, with no further UI information like containment among widgets, layout [43]. This is meant to be this way since mockups were conceived as quick-to-build requirements artifacts and not as software specifications; thus, their structure is kept as simple as possible to accomplish their objective of communicating and persisting requirements. On the other hand, HTML prototypes are too detailed since their structure is inherently complex and they have plenty of elements (their internal DOM) that are not necessarily related to data requirements specifications. Having a well-defined UI structure where the important elements (from the data modeling point of view) are detailed and highlighted can help to detect and build data models from interface prototypes, as described in [23].

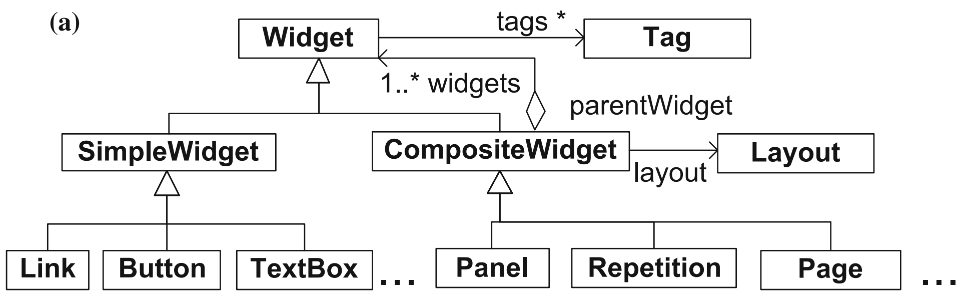
For the purpose of formalizing the UI structure independently of the type of mockup used, we generate an instance of a *Structural User Interface* (SUI) model that is mapped over the mockups. The SUI model is a User Interface Description Language (UIDL) proposed in [14], i.e., a formal language that allows to describe the structure and composition of a User Interface with a specific level of detail and in a platform-independent way. It is composed by a set of simple and composite widgets and provides a basic structure to UIs represented using mockups, i.e., it complements mockups with information about the structure of the UI they represent. A graphical representation of the SUI metamodel can be observed in Fig. 2a, while an example of a SUI instance associated to an HTML mockup is shown in Fig. 2b. Although adding a SUI model is not mandatory during the modeling tasks (by default every mockup is converted to a Screen, without any internal component—see Fig. 2a), having the detailed structure of the UI in the form of a SUI instance can help to detect modeling errors, as we will show later. Finally, the introduction of the SUI over a UI mockup helps to accomplish a detailed traceability: every UI component that is important from the data modeling point of view is highlighted and referenced in mockups, which are the main requirement artifacts used in our methodology.

While several User Interface Description Languages (UIDLs) currently exist (with UsiXML [44] being one of them), we decided to use our custom UIDL (i.e., SUI) for the following main reasons: (1) it is meant to describe aspects of UI mockups that are relevant to data requirements specification, (2) it was designed to be enriched with annotations (tags) and (3) to maintain compatibility with our previous work. On the other hand, while a detailed comparison between SUI and other UIDLs is out of the scope of this paper, the SUI structure is compatible with widget-oriented UIDLs such as the UsiXML CUI model.

The tooling implemented for DataMock supports all the three types of mockups mentioned above: bitmaps, tool-based, and HTML. When bitmap mockups are used, no SUI can be generated. When using tool-based mockups, a first version of the SUI can be generated using the previously introduced MPE, whose technical details can be found in [43]. Finally, if HTML mockups are used, developers have to identify which parts of the UI structure are relevant from the data modeling point of view—which will form the final SUI. This identification is accomplished interactively using the ITT tool: Through DOM manipulation of the original HTML mockup, the tool allows to highlight and select elements that will form the SUI model. Every time the user selects an element of the mockup, an instance of the SUI Widget subclass is created and made persistent. Following this step-by-step strategy, the SUI model is created iteratively as the user selects relevant elements in the mockup. More technical details of this approach are provided in Sect. 3.5. The different artifacts and tools involved in the process are graphically represented in Fig. 1.

3.2 Step 3: Mockups Annotation

The central task of the DataMock method is the annotation of mockups with data specifications. Annotations can be placed in any order and over the different types of mockups introduced earlier. As said before, at any point of the iterative data modeling process annotations can be used to generate the current data model in the form of UML Class Diagram. Mockups are annotated with *tags* [17] that can be formally applied over specific UI widgets (if a SUI model is defined) or just freely in the mockup (if no SUI is available, e.g., when bitmap mockups are used). In the latter case, tags can be placed graphically over parts of the mockups to preserve traceability, but they are not formally associated to SUI widgets. As



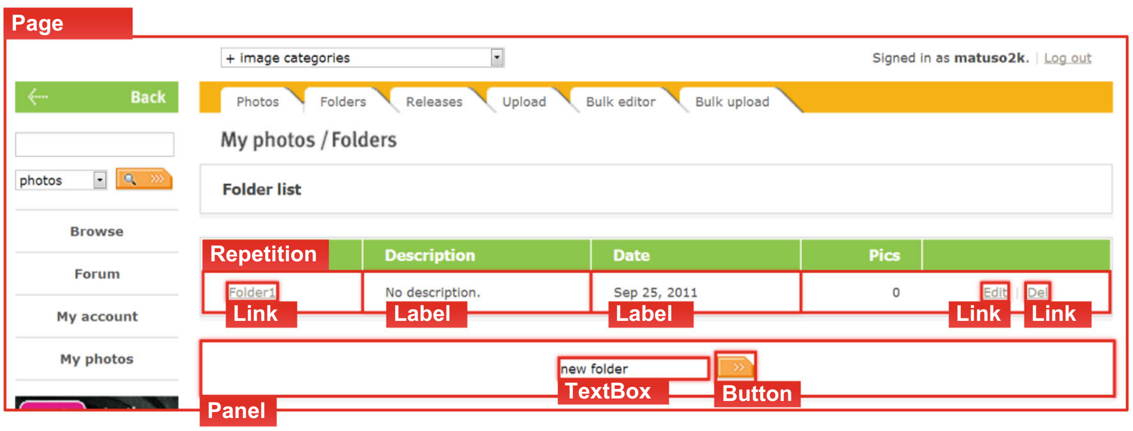


Fig. 2 The SUI metamodel and an example of its application: **a** Main components of the SUI metamodel, **b** SUI model mapped over a HTML mockup of a photo stock Web application

a common practice when prototyping, different copies of the same mockup are used to represent the dynamic behavior of a given UI in which some widgets could appear, disappear, or move because of the user interaction and the UI changing its state [43]. These copies are *wired* using navigation features provided by mockup tools. Using this practice, DataMock can also be applied in contexts of applications with dynamic UIs since a concrete set of widgets will exist for every state, and tags can be applied over all of them.

However, as the number of mockups to draw/copy/annotate increases with the number of UI states to represent, the method poses some limitations when dealing with applications with an extremely dynamic and very complex UI, unless only a minority of UI states are worthy to be presented (and thus modeled) to the stakeholders.

Tags are textual descriptions composed by a name and a set of parameters following the form `TagName(param1, ..., paramN)`. The `TagName` identifies the type of feature that is being modeled over the mockup. Each tag type has its custom set of parameters. Each parameter has its degree of complexity, varying from a simple text value to an expression in a custom Domain-Specific Language (DSL). Since we are making emphasis in data modeling, we will describe primarily the use of data-related tags, which have the `form Data(<data-expression>)`. In this case, instead of including only a class name (as we already introduced in [17]), *< data-expression >* is an expression in a custom DSL that can describe one or more data model specifications. The basic constructs of this language are shown in Table 2.

These basic constructs can be combined to form more complex data specifications. For instance, the following construct:

```
Data(Post => Publication.comments
  -> *Comment => Annotation)
```

denotes that a list of `Comments` is shown or manipulated in the UI. These `Comments` are obtained originally from a `Post`, traversing a `comments` association. At the same time, Post and Comment inherit from `Publication` and `Annotation` classes, respectively. Finally, the `comments` association is not part of the `Post` class itself, but belongs to its superclass, `Publication`. The syntax used in our `Data` tags reported in Table 2 can be seen very close to that of the Object Constraint Language[6] (OCL) syntax, specif-

[6] OCL—http://www.omg.org/spec/OCL/2.4/.

Table 2 Basic constructs of the DataMock's Data tags

Construct	Description
`Data([*]Class)`	Denotes that an object of class `Class` is shown or can be manipulated in the UI. If the optional * is used (e.g., `Data(*Post)`), it denotes that a list of such objects are shown or can be manipulated
`Data(Class.attribute[:datatype])`	Specifies that the attribute of an object of class `Class` (called `attribute`) is shown or can be edited through an underlying User Interface widget. Optionally, a data type can be defined for that attribute (one of `Date, String, Integer, Decimal, Boolean`, or `Blob`). If no datatype is specified, `String` is assumed
`Data(Class1.association -> [?][*]Class2)`	Denotes that an object of `Class2` is shown or can be manipulated through the underlying element in the UI. However, this element is obtained navigating from an association called `association` from another element of class `Class1`. If the * modifier is used, it implies that a list of `Class2` instances is shown or can be manipulated in the UI and that the association is one-to-many. On the other hand, if the ? modifier is present, it indicates that the association is not mandatory—i.e., there may not be any object referenced by the association
`Data(Subclass => Superclass)`	Denotes that an object of class `Subclass` is shown or can be manipulated in the User Interface and that the class of this object (`Subclass`) inherits from another one called `Superclass`

ically regarding navigation. However, OCL and DataMock tags have different focuses: while OCL is meant to be used over existing UML models to specify constraints that cannot be expressed using just pure UML constructs, DataMock tags are intended to declare the data model itself using navigations. That implies that navigations in `Data` tags allows to include details like minimum and maximum cardinality and inheritance, features that OCL does not allow to declare. In fact, OCL is written over an existing data model and does not to add new elements to that but only constraints to be fulfilled.

The previous example shows the semantic power of the DataMock `Data` tags; in this case, in one tag, seven different data model elements are defined: four classes, two inheritance relationships, and one association. While it is true that developers will rarely define such complex tags at once in a single iteration, it is expected for these to appear after several iterations, when the development team gains a clearer vision of the domain objects. Example of a tagged tool-based mockup is depicted in Fig. 3.

Every tag applied over a mockup represents the potential addition of one or more data model features. These features could define completely new data model elements (for instance, new classes) or features related to existing ones. Data features in tags are idempotent—i.e., the application of the same specification several times has no effect after the interpretation of the first one. Extending the example of Fig. 3, if a new invoice listing mockup was tagged with `Data(*Invoice)`—i.e., a list of `Invoices` will be shown in it—no extra class will be created since the `Invoice` class has already been specified in another mockup.

In Fig. 4, the automatically generated data model expressed in the form of a UML Class Diagram and its source tags are shown. It is important to note that every data specification (tag) in DataMock is traceable, since it is explicitly related to a mockup concept. Since data models are generated from these tags, it implies that the final data models are also completely traceable. In the particular case of Fig. 4, the mockup was created using the Pencil tool and the tags were placed directly over the mockup using the annotations features already present in the tool—thus, no SUI model was defined.

To exemplify how the iterative data model building works from the operational point of view, assuming that the tags are processed from top to bottom as placed in Fig. 4, the following steps are executed:

1. New mockup to be processed is found. An empty UML Class Model is created.
2. `Data(Invoice)` tag is found. `Invoice` class is created since it does not exist yet in the model.

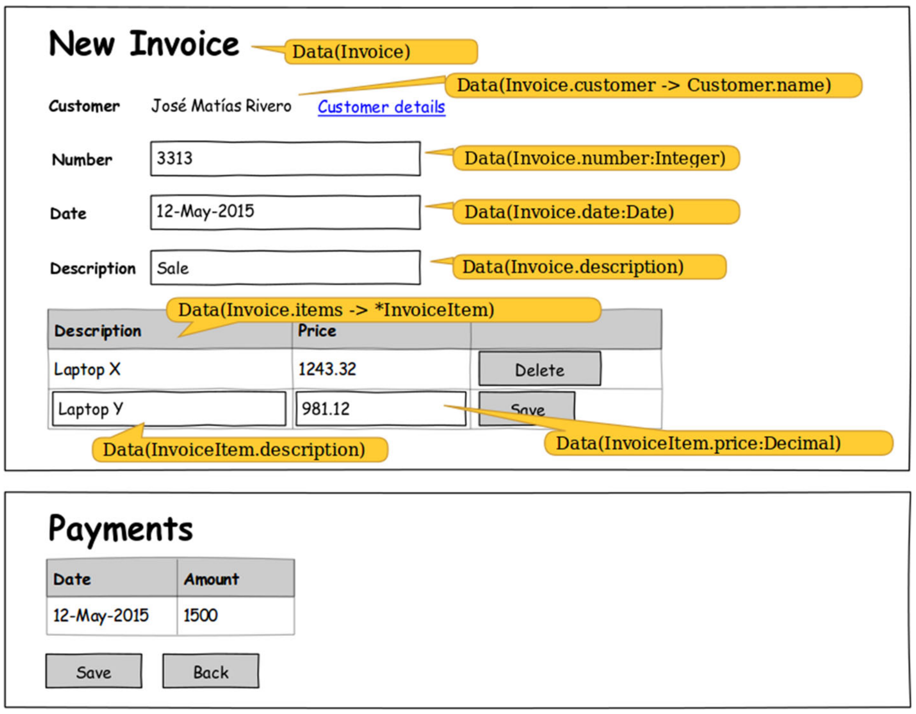


Fig. 3 A tagged mockup example

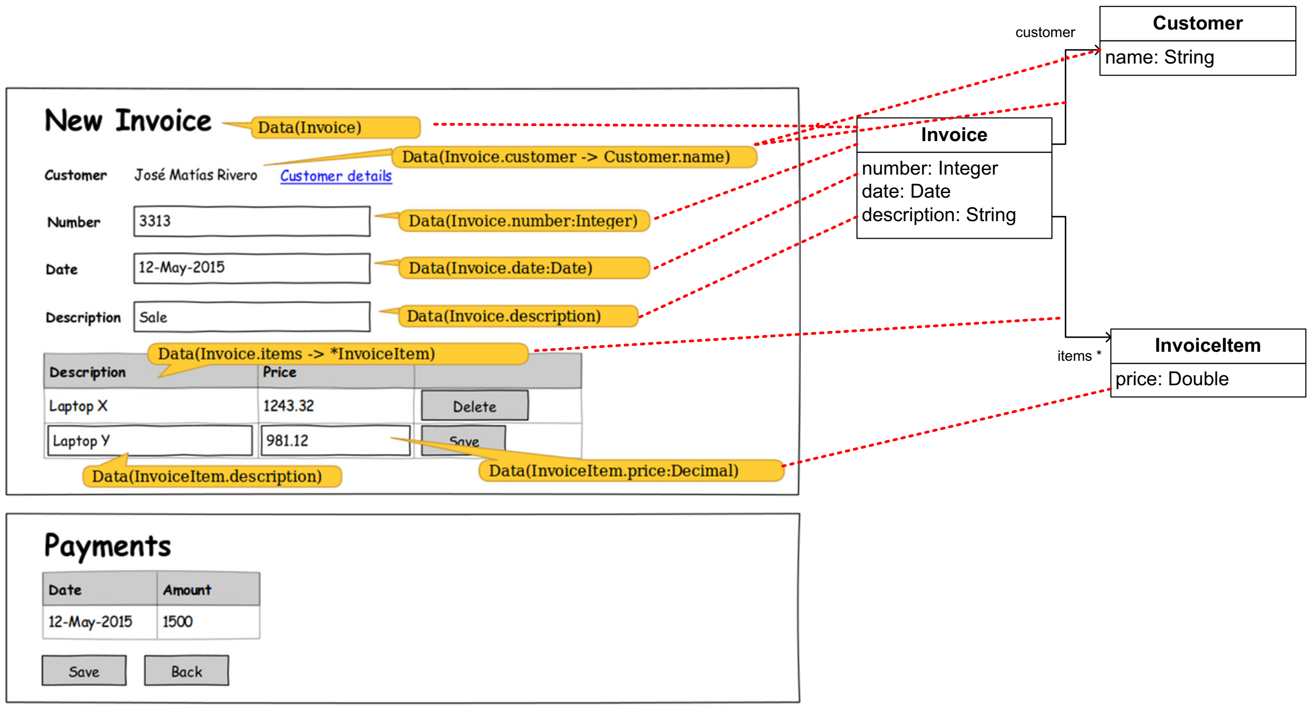


Fig. 4 A tagged mockup and the data model generated from it through tags processing

3. `Data(Invoice.customer− >Customer.name)` tag is found. `Customer` class is created, since it does not exist. name property in `Customer` class is created since it does not exist, `String` data type is assumed by default since no data type was explicitly specified. Finally, the `customer` association from `Invoice` to `Customer` is created, since it does not exist.

Table 3 Equivalence between the developer- and end-user-oriented textual representation of the metamodel

Developer-Oriented Grammar	End-User Oriented Grammar
`Data(Invoice)`	`an invoice`
`Data(*Invoice)`	`a list of invoice`
`Data(Invoice.number:Decimal`	`an invoice has a number, which is a decimal value`
`Data(Invoice.lines -> *InvoiceLine)`	`an invoice has a list of invoice lines associated, called "lines"`
`Data(Invoice.lines -> *InvoiceLine.product -> ?StockProduct => Product.name)`	`an invoice has a list of invoice lines associated, called "lines"; an invoice line has an optional stock product associated called "product"; a stock product is a type of product; a product has a name;`

4. `Data(Invoice.number: Integer)` tag is found, `number` property (with `Integer` data type) is created in `Invoice` class since it does not exist.
5. `Data(Invoice.date: Date)` tag is found, `date` property (with `Date` data type) is created in `Invoice` class since it does not exist.
6. `Data(Invoice.description)` tag is found, `description` property in `Invoice` class is created since it does not exist, `String` data type is assumed by default since no data type was explicitly specified.
7. `Data(Invoice.items−>∗InvoiceItem)` tag is found, `InvoiceItem` class is created, since it does not exist. `items` (one-to-many) association from `Invoice` to `InvoiceItem` is created, since it does not exist.
8. `Data(InvoiceItem.price: Decimal)` tag is found, `price` property (with `Decimal` data type) is created in `InvoiceItem` class since it does not exist.
9. `Data(InvoiceItem.description)` tag is found, `description` property in `InvoiceItem` class is created since it does not exist, `String` data type is assumed by default since no data type was explicitly specified.

Tags introduced so far are clearly modeler-centered, since they include implicitly technical modeling concepts such as classes, properties, inheritance, associations. However, in order to facilitate involving end-users in the modeling process, DataMock also provides a more verbose but end-user friendly way of composing tags using the natural language. For instance, `Data(*Invoice)` can be also referred as `a list of invoice`, and `Data(Invoice.number: Integer)` is equivalent to `an invoice has a number, which is a numerical value`. The former grammar is called Developer-Oriented Grammar (DOG) and the latter End-user-Oriented Grammar (EOG). Using EOG, end-users can then understand data tags more clearly, thus facilitating their involvement in the data modeling process. From the technical point of view, this is accomplished using a metamodel that abstracts tags from this inherently textual representation. A more detailed description of the equivalences between the two grammars is provided in Table 3. Currently, the mockup processing tooling (described later) accepts both languages, allowing developers to choose which suits better for the modeling task. Since both textual syntaxes are supported, the tool also allows converting from one form to the other—i.e., one grammar is fully convertible and traceable to the other. Note that some implicit rules are used when converting from one grammar to another—for instance, camel-cased class names in DOG are translated into separate words in EOG and vice versa.

In addition to the Data tags (which follow the form `Data(<data−expression>)`, in the DataMock tags language we also included simple navigation specification tags. These tags allow specifying additional information that can be useful to suggest data model refactorings, as we will show later in the paper. Navigation tags follow the form `Link(<mockup−name>)`, where `<mockup−name>` is the name or *id* that identifies a mockup unequivocally.

In order to obtain a syntax-independent tag representation, we defined a tag metamodel, called Spec Metamodel (SM), as depicted in Fig. 5. The metamodel allows to represent Class Diagram concepts in simple terms, thus being semantically enough to generate them. The core class in the Spec Metamodel is `DataSpec`. A `DataSpec` ref-

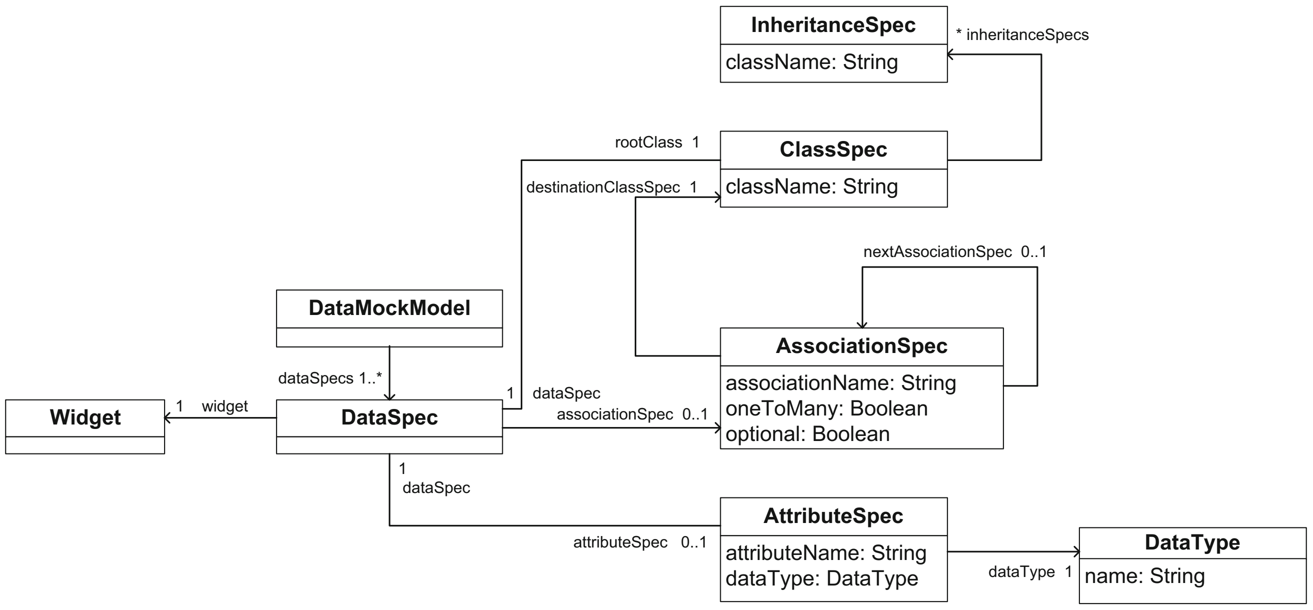


Fig. 5 DataMock Spec Metamodel

erences at least a `ClassSpec` class or class hierarchy through a `rootClass` association. A `ClassSpec` has a `className` attribute and a list of `InhertianceSpecs` that may define a hierarchy of classes to represent tags like `Data(Class=>Superclass1=>Superclass2)`. A `DataSpec` can also define an attribute and its data type through `AttributeSpec` to allow representing tags like `Data(Class=>Superclass1=>Superclass2.superclassAttribute : String)`. It may also define one or more associations through a chained list of `AssociationSpecs`, thus allowing to represent tags like `Data(Class=>Superclass.associationName − >AssociatedClass.attribute : String)`. An `AssociationSpec` has a name (`associationName`), a destination `ClassSpec`, and a minimum and maximum cardinality (`oneToMany` and `optional` attributes).

It is important to note that since SMs are a way of representing data tags in a syntax-independent way, several elements in the SMs can generate a single element in the resulting Class Diagram. For instance, if several `Data(Invoice)` tags are placed in different mockups related to an application being modeled, several `ClassSpecs` with `className = "Invoice"` will be used to represent these tags, but only one UML class will be generated to represent the class `Invoice`. However, from the semantic point of view, the following correspondences can be established between elements in SM and UML Class Diagram (CD)[7] primitives:

– `Classes` in CD are represented by `ClassSpecs` in SM
– `Generalizations` in CD are represented by `InheritanceSpecs` in SM
– `Associations` in CD are represented by `AssociationSpecs` in SM
– `Property` elements in CDs are represented by `AttributeSpecs` in SM

Having the SM syntax-free form of representing tags, converting from EOG to DOG syntax and vice versa is accomplished by a *parser* that converts the original representation to an instance of the SM and then by a *renderer* that transforms the SM instance into the destination representation.

3.3 Step 4: Mockups and Tagging Processing and Data Model Generation

The data model generation process is detailed in Listing 1. The first step of the process consists in *merging* the individual data specifications defined by each tag (Listing 1, lines 5 to 10). The process starts by parsing the tags placed on mockups to create an instance of the SM (Listing 1, lines 7 to 8). After obtaining such an instance and starting from an empty Class Diagram, the Model Generator tool iterates over every element of the SM instance (Listing 1, lines 11 to 22) and adds a corresponding data model element in the UML Class Diagram if not defined yet (Listing 1, lines 13 to 15). When any inconsistency or error in the tags or the merged model being built is found, an error is registered in the model generation report and the involved elements are omitted until modelers decide what to do with them (Listing 1, lines 17 to 20). The

[7] OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF.

output of this process is the model increment for the current iteration of the process and a list of the errors or warnings found (if any) during the model generation, as described by the current data specification tags in the mockups (Listing 1, line 23).

At this point it is important to note that the DataMock proposal is focused on data requirements and persistence and is restricted to classes of the data model of an application that can be visually described by end-users through mockups. Our proposal can also be used to model the boundary or control-related classes of an application too (for instance, classes denoting operations to be executed or showing a report in the UI which is a summary of existing information in the system), but this possibility is out of the scope of this paper. If a developer or analyst includes these classes using the provided annotation set, they will be included in the generated UML Class Diagram models, which is not strictly wrong, since they are part of the system as well. However, the future use or role that will have in the system (persistence, control, information summarizing, etc.) is out of the scope of this work.

DataMock can create duplicated classes due to typos, e.g., `Invoice` versus `Inovice`. On the other hand, since the approach allows to tag the same concepts in different mockups (when they appear more than once), it also allows to unify definitions and detect possible data modeling conflicts—for instance, a property represented as a `String` in one tag and as an `Integer` in a different one. In addition to the basic model inconsistency detection included in Listing 1, the DataMock tooling implementation includes a set of rules and heuristics that are applied during the model transformation to detect and solve some of these errors or inconsistencies in order to obtain better data models. In the same sense, processors within the tool can detect errors introduced because of bad modeling, not necessarily related to the DataMock procedure. We call both types of errors *fail patterns* (analogous to bad smells in traditional refactoring literature [45]) and provide one or more solution for each one—called *tag refactorings*.

Some examples of these patterns are shown in Table 4. In particular, Listing 2 reports the algorithm for the detection

```
1.  processMockups(mockups: Mockup[])
2.    dataModel <- new Class Diagram
3.    errorList <- empty Error list
4.    specModel <- new Spec Model
5.    for each mockup in mockups
6.      for each tag in mockup.tags
7.        dataSpecs <- parse tag and obtain DataSpec(s)
8.        add dataSpecs to specModel
9.      end for
10.   end for
11.   for each ds in specModel
12.     element <- determine data model element from ds
13.     if dataModelElement not exists in dataModel
14.       add dataModelElement to dataModel
15.     else
16.       existingElement <- get existing element from model
17.       if existingElement <> ds
18.         error <- get error description from inconsistent elements
19.         add error to errorList
20.       end if
21.     end if
22.   end for
23.   return dataModel and errorList
24. end
```
Listing 1. Pseudocode of mockups and tagging processing

3.4 Assisted Error Detection

The modeling approach that DataMock proposes is *sparser* in the sense that it specifies data models elements using relatively isolated specs applied in an iterative way. While this characteristic allows explicitly validating and tracing data requirements, it can introduce errors that are not common when building data models in one step. For instance,

of the repeated attributes error. The first step in the algorithm scans each DataSpec instance in the DataMock model which contains a class, a potential property and data type, and one or more inheritance relationships in chain and (1) registers the class if it does not exist yet (Listing 2, line 8) and, if an attribute is defined in that DataSpec, (2) determines the root class to which it is applied to; after this root class is found, it is added to the corresponding class (Listing 2, lines 12 to 17). In this first step, we also register and index inheritance

Table 4 Partial basic fail pattern list supported by DataMock tooling

Pattern	Description, example and solution

Name Typos	• *Description*: Typos in tags can lead to different classes, attributes or associations that should be unified.
	• *OCL invariant rule for detection (for attributes, the same applies for classes or associations):*
	`context AttributeSpec inv:`
	`not AttributeSpec.allInstances -> exists(as |`
	`  as <> self and`
	`  leventstheinDistance(`
	`    as.associationName,`
	`    self.associationName) <= 2`
	`)`
	• *Example*: five `Data(Invoice)` tags are found and only one `Data(Inovice)` – typo – in mockups.
	• *Solution*: Group class, attribute, or association names with low string distance – e.g., using the Leventsthein algorithm [46]. The most used name will be proposed to *correct* the less used names in the tags.

Repeated Attribute in Hierarchy	• *Description*: After building the data model, attributes with the same name and type exist in different classes within a class hierarchy. • *Detection pseudocode: included in Listing 2.* • *Example*: `Data(Invoice.number:Integer)` and `Data(Invoice => Document.number:Integer)` tags are defined. The `number` attribute is repeated in `Invoice` and in its superclass, `Document`. • *Solution*: Remove attribute in the subclasses.

Inconsistent Data type	• *Description*: The same attribute or association has been defined in the same class and with the same name, but with different type or destination class respectively.
	• *OCL invariant rule for detection:*
	`context AttributeSpec inv:`
	`not AttributeSpec.allInstances -> exists(as |`
	`  as <> self and`
	`  as.attributeName = self.attributeName and`
	`  as.dataType <> self.dataType`
	`)`
	• *Example*: `Data(Invoice.number:String)` and `Data(Invoice.number:Integer)` tags are defined.
	• *Solutions*:
	  ○ Select one type and unify attribute types to use a unique type – propose the most used type as default.
	  ○ Rename one of the attributes so they can coexist.

relationships between classes individually (Listing 2, lines 20 to 24). Since in a DataSpec instance several inheritance relationship specs can be defined in a chain, this eases further processing. Once all classes, attributes, and inheritance relationships are indexed, the inconsistency detection step starts (Listing 2, lines 27 to 35). In this step, all the attributes indexed for all the classes are scanned (Listing 2, lines 27 and 28) and, for each one, the full hierarchy is traversed through the `attributeFoundInHierarchy` method (Listing 2, lines 41 to 48) to check whether an attribute with the same name exists. If such attribute is found, the process ends and a message error is returned with the attribute and class in which it is repeated. If no attribute matching this condition is found, no error is thrown and the process ends gracefully.

While the DataMock tooling is capable of detecting and applying semiautomated solutions by solely analyzing the tags list placed over the mockups as in the previous examples, having a correctly mapped SUI model over the mockups can help to detect more complex modeling flaws and solve them. Two examples of patterns that make use of the SUI model are shown in Table 5. Note that, when the SUI model is mapped, these rules make use of the widget property present in the DataSpec instance—when SUI is not available, this property will be set to `null` value.

```
1.  repeatedAttributeInHierarchy(model: DataMockModel) {
2.    Map<String, String> inheritanceMap = new Map<String, String>();
3.    Map<String, Set<String>> attributeMap =
4.      new Map<String, Set<String>>();
5.
6.    // builds inheritance map
7.    for(DataSpec ds in model.dataSpecs) {
8.      attributeMap.putIfKeyDoesNotExist(class, new Set<String>())
9.      // register attribute if defined
10.     if (ds.attributeSpec != null) {
11.       // get final class
12.       if (ds.inheritanceSpecs.nonEmpty()) {
13.         String class = ds.inheritanceSpecs.last().className
14.       } else {
15.         String class = ds.rootClass.className;
16.       }
17.       attributeMap.get(class).add(ds.attributeSpec.attributeName)
18.     }
19.     // register inheritance relationships in spec
20.     String class = ds.rootClass.className;
21.     for (InheritanceSpec is in ds.rootClass.inheritanceSpecs) {
22.       inheritanceMap.put(class, is.className);
23.       class = is.className;
24.     }
25.   }
26.   // seeks for repeated attributes in hierarchy
27.   for (String class in attributeMap.keys) {
28.     for (String attr in attributeMap.get(class)) {
29.       if (attributeFoundInHierarchy(class, attr, inheritanceMap,
30.                                     attributeMap) {
31.         return "Attribute " + attr + " repeated in class " + class;
32.       }
33.
34.     }
35.   }
36. }
37.
38. attributeFoundInHierarchy(String class, String attribute,
39.                           Map<String, String> inheritanceMap,
40.                           Map<String, Set<String>> attributeMap)
41.   while (class != null) {
42.       if (attributeMap.get(class).contains(attribute)) {
43.         return true;
44.       } else {
45.         return false;
46.       }
47.       class = inheritanceMap.get(class);
48.   }
49. }
```

Listing 2. Java-like of Repeated Attribute in Hierarchy detection algorithm

Table 5 Partial fail pattern list that requires a mapped SUI model

Pattern	Description

Isolated contained class to association	• *Description/Detection*: A `CompositeWidget` has a class-related tag (for instance, following the form `Data(Class)` or `Data(Class1.attribute -> Class2).` An inner `CompositeWidget` or `SimpleWidget` references a different class without referencing the one in the parent `CompositeWidget`. Because of the containment relationship expressed in the SUI, it is highly expected that the tag belonging to the inner widget must reference the class mentioned on the container one through an association.
	• *OCL invariant rule for detection:*
	`context ClassSpec inv:`
	`not ClassSpec.allInstances -> exists(cs |`
	  `cs <> self and`
	  `cs.className <> self.className and`
	  `self.dataSpec.widget.parentWidget =`
	    `cs.dataSpec.widget`
	`)`
	• *Example*: A `CompositeWidget` has a `Data(Invoice)` tag. An inner `SimpleWidget` has a `Data(Customer.name)` tag. It is highly expected that some relationships must exist between `Invoice` and `Customer`; then, a tag refactoring from `Data(Customer.name)` to `Data(Invoice.customer -> Customer.name)` is suggested.
	• *Solution*: Add the class referenced in the container widget to the inner widget tag in a form of an association. Use the parent class name lowercased as a default name for it.

Implicit Association	• *Description/Detection*: Two different classes appear in different tags of the same mockup (at a root level, for instance, associated to `CompositeWidgets`) or in different mockups linked through a navigation (`Link()`) tag. However, no explicit relationship is defined so far between these classes.
	• *OCL invariant rule for detection:*
	`context ClassSpec inv:`
	`not ClassSpec.allInstances -> exists(cs |`
	  `cs <> self and`
	  `cs.className <> self.className and`
	  `self.dataSpec.widget.parentWidget =`
	    `cs.dataSpec.widget.parentWidget`
	`)`
	• *Example*: `Data(*Invoice)` and `Data(Customer)` tags appear in two different mockups. They are also related through a navigation tag that connects a mockup containing a `Data(Customer)` tag to another that has a `Data(*Invoice)` tag. However, no data tag implies an association between both, which seems the case, because of their *visual* and *functional proximity*.
	• *Solution*: Create and association between both classes. The direction, cardinality and name of this association are suggested from the tags to be related, depending on their cardinalities or the existence of a navigation tag. If both classes are present in one mockup, the user has to choose the direction of the association – i.e., over which class it should be defined. On the other hand, if a navigation tag is present, the class belonging to the source mockup will define it. In both cases, the lowercased name of the destination class is used as an initial name for the association.

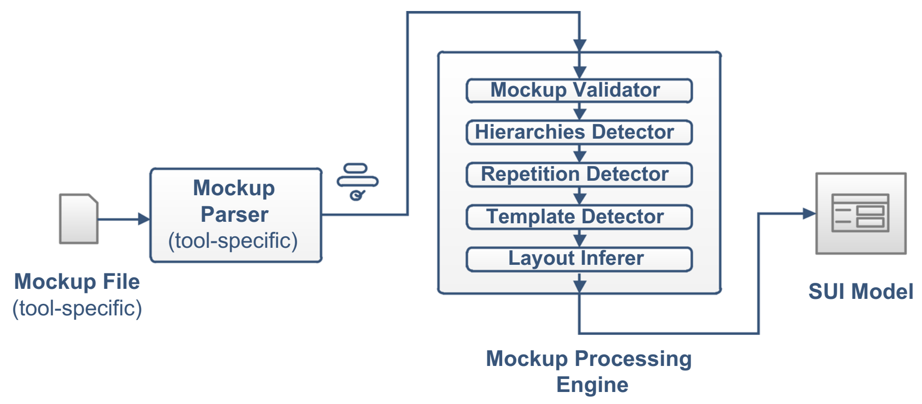


Fig. 6 Mockup Processor Engine processing workflow and components

3.5 DataMock and Its Supporting Tools

In this section, we briefly describe the tool support available for the DataMock approach. As depicted in Fig. 1, there are four tools involved in the data modeling process. The first one corresponds to *mockups tools*, i.e., any tool that can be used to create mockups, such as Balsamiq or Pencil. Mockup tools allow creating digital mockups with low effort in comparison with traditional IDEs or WYSIWYG tools frequently used by developers. As commented earlier, while mockup tools make mockup drawing a very simple and quick task, they lead to poor UI specifications from the point of view of software requirements specification. If tool-based mockups are used in DataMock the *Mockup Process Engine* (MPE) tool allows processing them and obtaining an instance of the SUI model [17,43].

The MPE tool (whose structure is depicted in Fig. 6) defines a parser for every type of tool-based mockup supported—for instance, Pencil or Balsamiq files. This parser returns a list of individual SUI widgets without any order or containment relationship. This widget list is transformed and validated by a pipeline of individual processors. First, a *Mockup Validator* checks that the widgets graphical disposition is valid—i.e., containment widgets are in fact `CompositeWidgets` and there is no visual collision between `SimpleWidgets`. Then, a *Hierarchies Detector* analyzes the graphical containment relationship between widgets and determines which `Widgets` are contained in which `CompositeWidgets`, if any. After this step, a *Repetition Detector* detects widgets that have been duplicated by *copy-paste* and transforms them into `Repetitions`. Finally, a *Layout Inferer* is used to choose a specific `Layout` for internal widgets in `CompositeWidgets` and their internal disposition according to the layout used. Details about the Layout Inferer and the Repetition and Template Detectors are omitted in this work since not relevant from the data modeling point of view. All these processors have inference parameters or arguments that can be tuned accordingly—for instance, the *Repetition Detector* defines a distance threshold between groups of widgets in order to determine whether they are part of a `Repetition` or whether they are individual widgets.

If HTML mockups are used, the *Interactive Tagging Tool* (ITT) allows importing them and defining which DOM elements are relevant from the data modeling point of view, thus building the SUI interactively. The ITT tool (depicted in Fig. 7) consists of a Web application that facilitates importing HTML mockups with their required internal resources like images or stylesheets through compressed files. Then, it shows the original mockups enriched with client-side scripts (i.e., JavaScript scripts) and allows interactively highlighting and selecting the DOM elements that will form part of the SUI. On the other hand, the server-side of the tool implements several REST endpoints that allow the client-side to inform new elements marked in the client-side and composes the SUI on the server, accordingly.

Tagging can be accomplished in different ways. If tool-based mockups are used, tags can be placed using commenting features already present in mockup tools. In this case, tags are associated to `Widgets` when being parsed by the corresponding *Mockup Parser* within the MPE. On the other hand, when mockups are imported in the ITT tool, tags can be placed over the marked elements that form the SUI (see Fig. 7). In both cases, the result is a SUI in which every `Widget` can have a tag associated. It is important to note that, since the SUI is optional, annotation features in mockups tools can also be used to only to place tags freely in a *non-formal* mockup—for instance, a bitmap image embedded in the mockup file. In this case, the MPE tool just processes the tags and generates no widgets.

The last tool used in DataMock is the *Mockup Generator* (MG). This tool is triggered by the ITT tool and takes a list of tags (potentially associated to `Widgets`, if an SUI model was defined), and parses them to generate an instance of the Spec Metamodel. All the tags from all the mockups are considered in this process, which results in a complete and *merged* data model. The MG tool is triggered by ITT users, and if any semantic or syntactic error is found when parsing

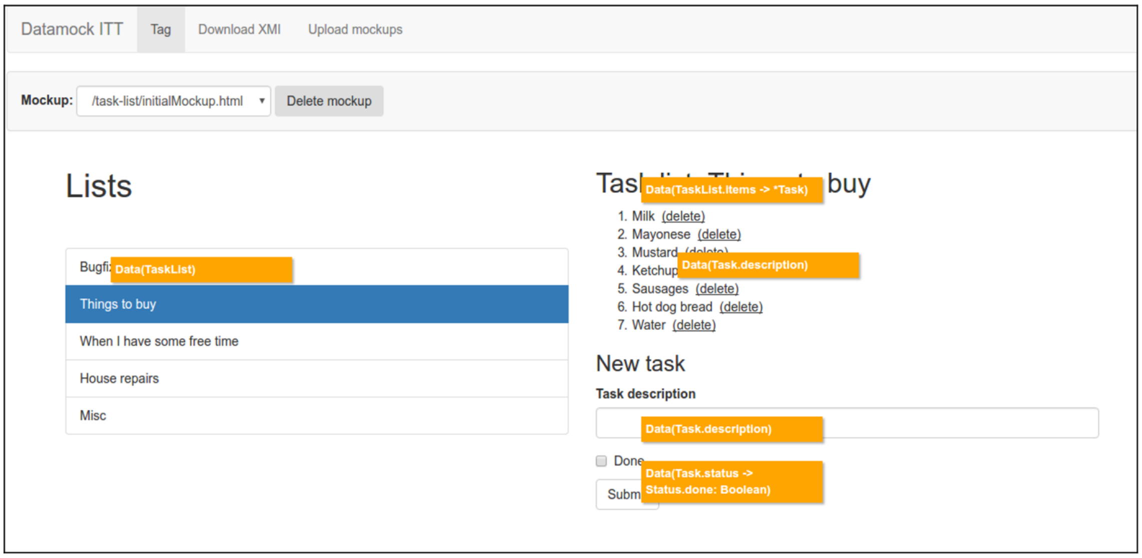


Fig. 7 ITT tool screenshot during an HTML mockup tagging

tags, it omits them and informs about the concrete errors in the web frontend of the ITT tool. Finally, the MG tool allows generating a data model in the form of a UML Class Diagram, using the XMI standard defined by the OMG.[8] This generation is accomplished through a concrete set of rules that transform every element of the Spec Metamodel instance to its textual XMI representation. Since the tool does not support data model visualization yet, generating models in this format allows to easily import and visualize them in tools like MagicDraw[9] or ArgoUML.[10] All this tooling is publicly available.[11]

4 A Full Example of DataMock Usage

So far we have introduced the technical and procedural aspects of DataMock. With this background in mind we will show, through an example, how requirements traceability is maintained within the approach. At this point we will continue with the invoice application example depicted in Fig. 4. The mockup shown in this figure was built using the Pencil mockup digital tool while the SUI model associated to it was generated from it using the MPE tool. Looking at the figure we can notice that there are parts of the mockup that are not tagged, i.e., all the interface elements of the *Payments* panel. From this visual hint and the validation of stakeholders, we can derive that there are parts of the mockup that cannot be traced to an underlying data model. After discovering this issue three new tags are added to the mockup corresponding to the untagged elements in Fig. 4:

- `*Payment`—a list of `Payments` are shown in the UI.
- `Payment.date:Date`—a payment has an attribute called `date`, which is of type `date`.
- `Payment.amount:Decimal`—a payment has an attribute `amount` which is a decimal number.

These additions will lead to the creation of a new class (`Payment`) with two attributes (`date` and `amount`). After invoking the MG tool through the ITT tool, a new data model with the new class and its two attributes is generated. However, the ITT tool detects a potential *Isolated contained class to association* fail pattern: As it can be noticed, in Fig. 7 two different `Panels`, tagged with different class tags, are sharing the same mockup, at the same hierarchical level, with no apparent relationship between them. Thus, the tool proposes a tag refactoring transforming the `Data(*Payment)` tag to $\mathtt{Data(Invoice.payment-> *Payment)}$ as a default (because of the cardinality of `Data(*Payment)` tag). In addition, the alternative `Data(Invoice)` to $\mathtt{Data(*Payment.invoice->Invoice)}$ transformation is also proposed as a second option. After choosing the default option, the resulting mockup and model generated from it can be seen in Fig. 8.

At this point, there is no data-related widget in the mockup without a tag (except for action widgets like buttons). This means that every date-related widget in the UI mockup has a corresponding tag that links it directly to one or more concepts in the data model. The three tags just added relate the widgets that were untagged to a class

[8] XML Metadata Interchange—http://www.omg.org/spec/XMI/.

[9] MagicDraw—http://www.nomagic.com/products/magicdraw.html.

[10] ArgoUML—http://argouml.tigris.org/.

[11] DataMock tooling source code repository—https://bitbucket.org/jmrivero/datamock-tool-public/.

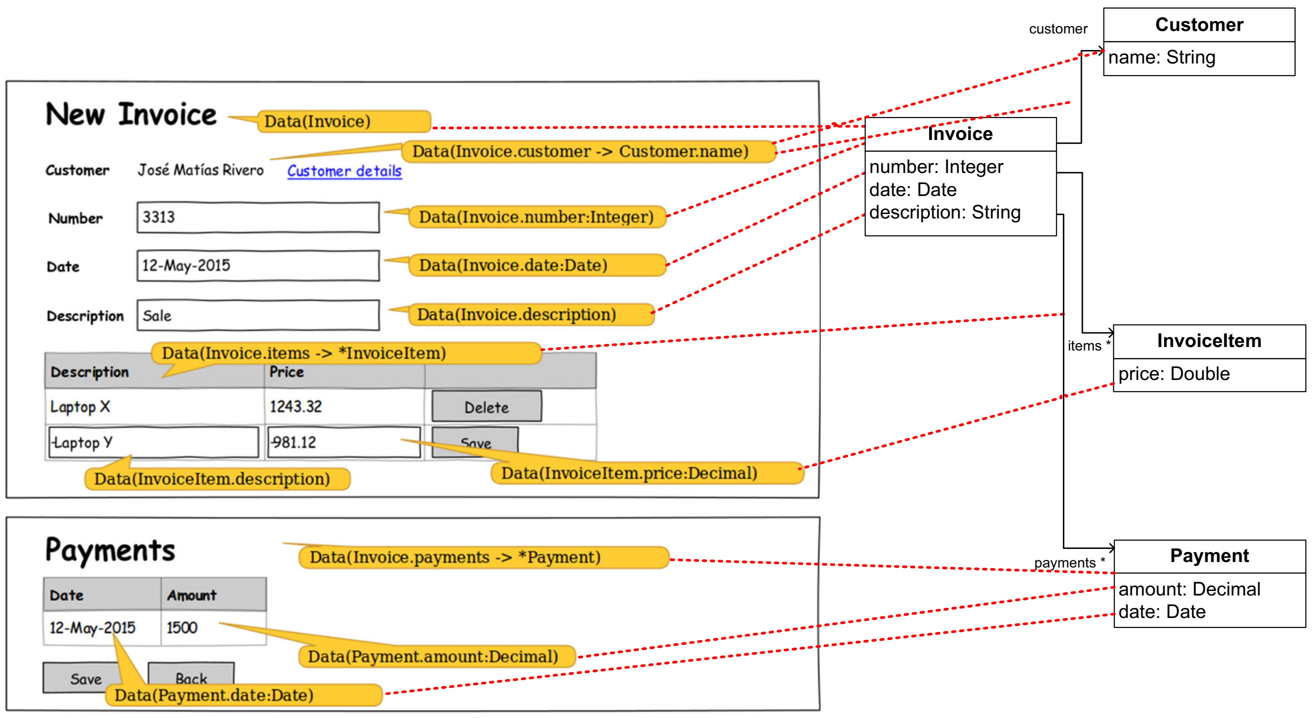


Fig. 8 Tagged mockup and resulting data model from tags, with traceability links between both

(`Payment`), two attributes (`date` and `amount`) and an association (`payments`) between `Invoice` and `Payment` in the data model.

Let's now assume that in a second iteration of the modeling process[12] the `InvoiceItem` price (total price) is decomposed and calculated from a unit price and a quantity of items ordered. This will lead to changes in the mockup: The table containing the products must change to reflect the new structure of items. Such updated mockup is depicted in Fig. 8. After the mockup is properly updated, two new attributes remain untagged in it, *unit price* and *quantity ordered*. Thus, the tags `Data(InvoiceItem.quantity: Integer)` and `Data(Invoiceitem.unitPrice: Decimal)` are added. At the same time, the attribute `price` is replaced by `totalPrice`.

In this second iteration, stakeholders may also propose to change the item description textbox to a dropdown menu, in order to choose from a subset of items. Selecting an item from the dropdown menu will imply that the unit price of such item is updated accordingly in the corresponding textbox. This change in the structure and interaction of the UI expressed by stakeholders denote that items are separated entities from the lines of the invoice since they should contain attributes like its description and unit price. To express this change in models, `InvoiceItem` is renamed to `InvoiceLine`, and `description` and `unitPrice` properties are *moved* to the Item class. This is expressed through the following changes in existing tags (Fig. 8):

`Data(Invoice.items− > ∗ InvoiceItem)` is changed to `Data(Invoice.lines− > ∗ InvoiceLines. item− >Item)`
`Data(InvoiceItem.description)` is changed to `Data(Item.description)`
`Data(InvoiceItem.unitPrice)` is changed to `Data(Item.unitPrice)`

After the tagging is finished, all the elements in the mockup are traceable to the final data model that can be generated through the MG tool, as can be seen in Fig. 8. Since in this example we use tool-based mockups (built with Pencil) with embedded tags, the ITT tool cannot detect whether a tag was renamed or deleted and a new one was created in its place. This is the case of the older tag `Data(InvoiceItem.price)`, which was rewritten as `Data(InvoiceLine.totalPrice)`—implying both a class and attribute renaming. While this is not strictly relevant when regenerating the data model of an application yet to implement, it is important when dealing with changes in running environments—for instance, in a relational DB already implemented and being used. The reason behind this importance is related to data preservation: For

[12] The need for this second iteration might also come from an evolution of the developed application.

Table 6 Goal of the proposed experiment

Analyze the:	Delivered data models
For the purpose of:	Evaluating the DataMock approach
With respect to:	Data model productivity, error reduction, and requirements traceability provided by it
From the viewpoint of:	The project team
In the context of:	Development of different applications with simulated clients and users.

instance, while renaming an attribute/column in a running relational DB will preserve the existing data, deleting existing attributes/columns and creating new ones will delete the existing data. In fact, the problem of data storage structure maintainability over time can appear in any persistence technology that requires predefined data schemas (i.e., to define tables/entities/collections, their attributes/columns) to store data. To avoid this problem, before regenerating the models, the ITT detects deleted and new attributes by comparing the new model with the older one. Then, for every attribute, it asks the user whether it was effectively deleted or replaced by some of the new ones. If not, it is considered as deleted. On the other hand, if the user chooses a new attribute, it will be considered as an attribute renaming operation. The same operation applies for classes.

5 Assessing DataMock: A Controlled Experiment

In order to evaluate the concrete advantages that DataMock brings to the data modeling process in comparison with a traditional approach, we conducted a controlled experiment in which we measured the efficiency of the approach. We used the Goal-Question-Metric approach [47] to define the goal of the empirical study and to drive the experiment itself. Basically, GQM defines a certain goal, refines this goal into questions, and specifies metrics that should provide the information to answer these questions. A GQM goal template for the evaluation is described in Table 6. In the next subsections, we will describe in detail the evaluation study we conducted.

5.1 Planning Stage: Goal

The purpose of the evaluation is to make a quantitative and qualitative comparison of the DataMock method to build data models (concretely, UML Class Diagrams) versus a manual modeling approach that adopts mockups as a mere informational requirements artifact. We decided to compare our approach to manual modeling using mockups since, as we commented earlier in this paper, mockups are the most used requirements artifact in Agile methodologies which in turn are the software development methodologies currently most adopted in the industry [11]. The goal is to measure the potential advantages that DataMock can provide in terms of productivity and traceability in data modeling, in the context of a software development process, in particular, when using UML Class Diagrams as data models.

While providing full model generation and facilitating error detection (as DataMock does) can be seen as advantages in comparison with manual methods, this is not always the case. For instance, if the generated models are imprecise and require to be tuned, or if error detection produces too many false positives, it can take longer for a developer to generate a data model using DataMock than building them manually. For this reason, we decided to evaluate the DataMock modeling process as a whole, including the tooling and its model generation capabilities to validate its concrete advantages.

5.2 Definition Stage: Questions and Metrics

As required by the GQM method, we defined a set of questions to achieve the proposed goal. With this purpose in mind, the following set of questions was defined:

- *Question 1*: Does the DataMock approach allows building data models faster, in comparison with building them manually?
- *Question 2*: Does the DataMock approach allows building data models with less missing, misunderstood, or non-explicitly required elements than building them manually?
- *Question 3*: Does the DataMock approach allows building data models that are more traceable from the requirements point of view in comparison with models built manually?

To answer these questions, we defined a set of associated metrics, as the GQM method suggests. The metrics defined were the following:

- *Metric 1: Building Time (BT)*: The time taken to build a specific data model using a particular method (*manual* or DataMock).
- *Metric 2: Error Score (ES)*: A numeric *score* defining the impact of errors found in the generated data models using a particular method. To compute this score, we compare the models obtained as output of the DataMock and the manual approaches against an *ideal model* (called *control model*) built for every application by experts. Then, we check the kinds of elements that are missing, wrong, or not explicitly required. Each type of error has a score according to a predefined scale depending on its gravity.

Table 7 Error Scores used to compute the ES metric

Error type	Error Score	Description
Missing class	1	A class is missing
Missing attribute	0.5	An attribute of a class is missing
Missing association	0.5	An association between two classes is missing
Missing inheritance	0.5	An inheritance relationship between two classes has been omitted
Extra class	0.5	A non-required class has been included
Wrong data type	0.25	An attribute in a class has a wrong data type
Extra attribute/association	0.25	A non-required attribute or association has been included in a class
Extra inheritance	0.25	A non-required inheritance relationship has been added between two classes

The score for every error type is defined according to their conceptual importance in the data model.

- *Metric 3: Non-Traceable Elements (NTE)*: The amount of Non-Traceable Elements found in data models—i.e., the amount of data model elements that cannot be mapped to our main requirements artifacts: mockups.

As most well-known model quality frameworks do, we base our quality measure (ES) on a comparison between different elements [48]. However, in order to simplify the evaluation, we compute model quality by comparing models of the same type and structure. The artifacts required to compute the ES metric (the error scale and the control models) were specified and designed by a team of professors of the National University of La Plata with more than 7 years of experience in teaching and applying data modeling and object orientation. The models were built by the experts with no tool assistance, in order to avoid potential biases introduced by the tooling commented throughout this paper. Error types and their predefined scores are depicted in Table 7. To compare models against the control model, we made a semantic comparison—i.e., we matched the elements (classes, attributes, associations, etc.) according to their semantics in models, avoiding fine-grained design decisions like concrete classes or attributes names.

Following the GQM procedure, metrics were associated to questions in the following way: Metric 1 (BT) allows answering Question 1, Metric 2 (ES) facilitates answering Question 2, and Metric 3 (NTE) provides a concrete result for Question 3.

5.3 Data Collection Stage: Method

Participants To conduct our experiment, we randomly selected 30 bachelor students of the Computer Science degree of the National University of La Plata presenting the same advances level in the degree. Each participant was interviewed in order to verify whether they had knowledge of at least the basic concepts of data modeling and Object-Oriented Programming. We also checked that every student passed the basic data modeling courses within the degree and that all of them were able to build UML Class Diagrams using tools. As we will describe later, each participant worked with both approaches (manual modeling and DataMock). The group was composed by 18 men and 12 women, aged between 22 and 30 (mean 26.3), with 2–4 years (mean 2.3) of experience in data modeling. All of them were familiar with data modeling—i.e., they had taken and passed the courses that included concepts of data modeling using E-R diagrams and UML Class Diagrams.

Apparatus For manual modeling, we gave the students the freedom to use their favorite UML modeling tool for manually building the data models. On the other hand, the use of the Interactive Tagging Tool (ITT) was required for DataMock. We provided data tag stencils for common mockup tools (like, for instance, Pencil) and also provided an online version of the ITT. In addition, we introduced the general approach and the tooling to the participants in one training session 30 minutes long, so that they could start the experiment with full knowledge about both approaches (manual and DataMock). We chose to use data tag stencils and common tools in order to (1) use tools that were already familiar to participants, and (2) avoid possible biases introduced by the tooling implemented for DataMock, which was not entirely stable when the experiment was conducted.

Procedure. A set of *bitmap* mockups of three different applications were given to each participant—we will call them applications A, B, and C. The applications consisted in a Q&A site, a music catalog and search engine, and a movie streaming and discovering app, respectively. Descriptive statistics about mockups of the three applications can

Table 8 Descriptive statistics about mockups of the three applications

Application	Number of mockups	Average classes/attributes per mockup estimated by the experiment team
A (Q&A)	5	Classes: 4.4 Attributes/Associations: 7
B (Music catalog)	7	Classes: 1.85 Attributes/Associations: 3.22
C (Video streaming)	6	Classes: 2.5 Attributes/Associations: 2.33

be found in Table 8. Some of these data have been calculated and/or estimated by the experiment organization team. To simulate the development of state-of-the-art Web applications, the experiment organization team defined such applications based on existing Web sites (e.g., music portals or Q&A ones). Bitmaps mockups have been chosen to prevent participants changing their structure (something that can be done when using HTML or tool-based mockups) and also to provide them with the most detailed and faithful prototypes as possible. An average of five mockups per application were built, and every participant received the same set for the corresponding app. Every participant was asked to build data models taking these mockups as a foundational requirements artifact. The modeling was divided into two iterations per application. Two of these models had to be built using one particular approach (manual or DataMock), and the other approach had to be used for the remaining one. This distribution allowed us to avoid the potential bias introduced by a participant modeling the same application with both approaches, one after another—which is known as a *Maturation* validity threat [49]. Thus, every participant tested both modeling approaches but modeled every application only once, under one particular approach. The experiment team ensured that each application was modeled the same number of times with both modeling approaches. Finally, the order in which the applications had to be modeled was randomized. In order to consider a participatory design context, a part of the professors forming the experiment organization team acted as clients and end-users, solving possible doubts or ambiguities present in the mockups without using technical jargon—just talking in terms of business objects. These professors were involved in the mockups development and in the tagging stages. However, they were not implicated in the elaboration of the DataMock approach and they were not presented during the aforementioned training sessions. Finally, to obtain the full information set required, we also asked participants to carefully take note of the time spent in modeling under every approach and application.

Regarding measurement, the metric BT was measured using a stopwatch: Every time the participant started to model the stopwatch was started and then it was paused when he paused or finished modeling. The remaining metrics (ES and NTE) were computed manually by the experiment organization team, comparing models with each other and models with mockups. In particular, in the case of metric ES, the resulting model was compared against the control model in order to find differences that could indicate errors or non-optimal modeling. On the other hand, the NTE metric was computed comparing the resulting models with mockups, trying to map every concept modeled to a visual metaphor and finding which ones could not be mapped.

5.4 Interpretation Stage: Discussion of Results

To quantify the advantages of the application of DataMock (our treatment) versus manual data modeling using mockups, we computed the Cliff's Delta effect size metric [50]. We used Cliff's Delta as it is oriented to effect size measurements, it is nonparametric, and also it does not strictly require the assumption of normality in the data distribution. Cliff's Delta ranges between -1 and 1, with -1 indicating total failure (or success, depending on the semantics associated to the result) of the treatment and 1 indicating the contrary (e.g., the treatment performed better in the 100% of the cases).

For the BT metric, we computed the Cliff's Delta per application and approach using the modeling time measures taken by the modelers under both approaches. A box-plot of such time measures distribution can be observed in Fig. 9. The result of the computation returned values of -0.41, -0.51, and -0.91 for the applications A, B, and C, respectively. According to the Cliff's Delta semantics, through this metric we can strongly assess that DataMock modeling took less time in comparison with manual modeling. Since Question 1 was based only on this metric, we can answer it positively according to the results obtained.

As can be seen from the results, the building time improvement was clearly better in application C than in the other applications. We attribute that improvement to the fact that mockups in application C seems to be *less dense* (i.e., with fewer features per screen) than in applications A and B.

For the ES metric, with the error scale defined in Table 7, an Error Score was computed for every application and participant comparing its output model to the respective control model for the concrete application. The Error Score per application and participant consisted in the sum of the individual errors found when doing the comparison. A box-plot of such Error Score distribution can be observed in Fig. 10. Then, as in the BT case, we computed the Cliff's Delta per application to obtain the effect size of applying DataMock to the modeling process, in this case with regard to errors found in

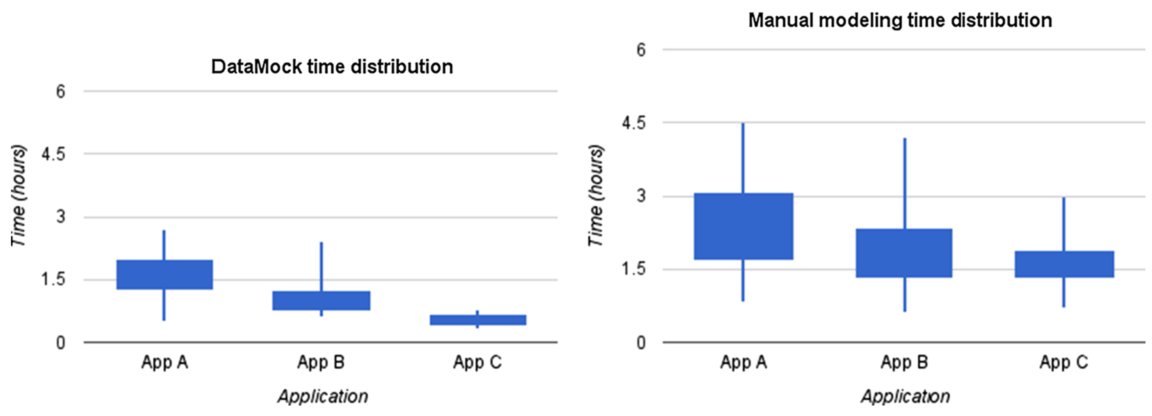


Fig. 9 DataMock versus manual data modeling time distribution

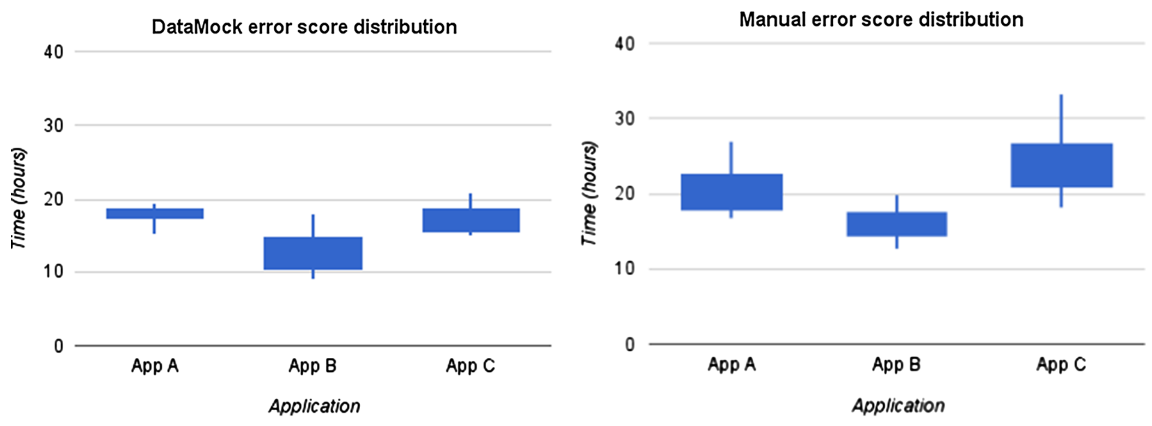


Fig. 10 DataMock versus manual data modeling fault error distribution

the resulting models. The result of the computation returned values of −0.12, −0.52, and − 0.86 for the applications A, B, and C, respectively. Again, according to the Cliff's Delta semantics, through this metric we can answer Question 2 positively and then strongly confirm that DataMock allowed building data models with fewer errors in comparison with building them manually. Also in this case the computation of the Cliff's Delta returned a better result for application C, and we infer that the main reason for this was the same that for the previous metric.

A similar approach to the ES case was used for the NTE metric (distribution shown in Fig. 11) whose computation returned the results −0.46, −0.7, and −0.53 for applications A, B, and C, respectively. Again, this metric allows answering Question 3 positively, as DataMock models shown to have a noticeable smaller number of untraceable elements compared to data models obtained manually.

In addition to the positive results provided by Cliff's Delta, we computed a statistical t-test for every metric to enforce the positive answers previously suggested by the effect size metrics. In this case, we decided to join the three samples obtained for every application into a single sample in order to increment the sample size for every test. Since the sample sizes for every application were the same for both methodologies, we assume that merging them does not represent a statistical bias in the test—because each application has the same *weight* in the sample, in both cases. The results were the following:

- For the BT metric, we concluded that, according to the given samples, the mean Building Time in the DataMock case is less than the mean in the manual modeling case with a confidence of 99% ($p = 0.006$, $\mu_{DM} = 1.11$, $\mu_M = 2.01$, $\sigma_{DM} = 0.71$, $\sigma_M = 1.00$)[13]
- For the ES metric, we concluded that, according to the given samples, the mean Error Score in the DataMock case is less than the mean in the manual modeling case with a confidence of 95% ($p = 0.029$, $\mu_{DM} = 16.38$, $\mu_M = 19.99$, $\sigma_{DM} = 2.95$, $\sigma_M = 4.84$).
- For the NTE metric, we concluded that, according to the given samples, the mean of Non-Traceable Elements in the DataMock case is less than the mean in the manual

[13] μ_{DM} and μ_M stand for μ values for DataMock and Manual modeling, respectively; σ_{DM} and σ_M stands for σ values for DataMock and Manual modeling, respectively.

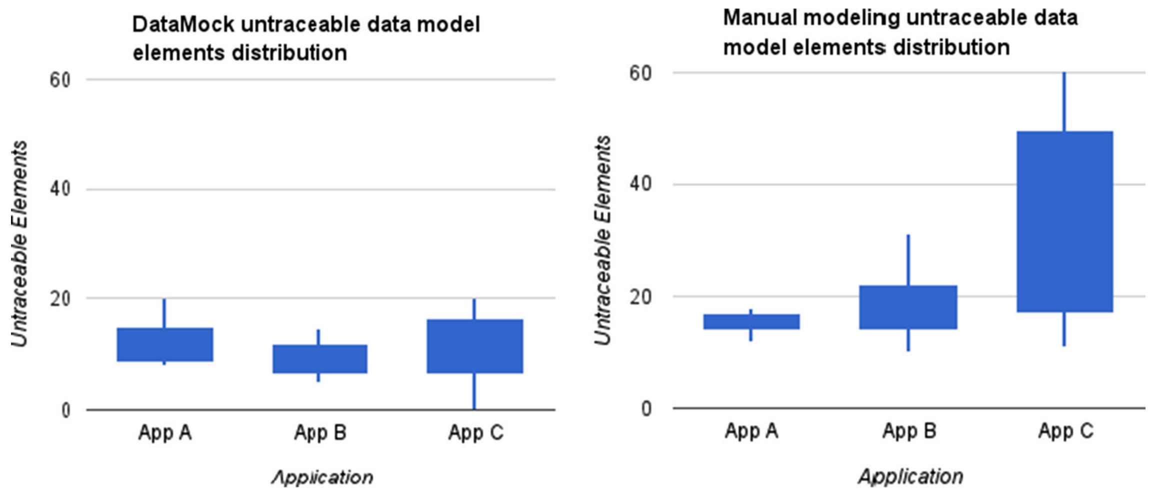


Fig. 11 DataMock versus manual untraceable data model elements

modeling case with a confidence of 99% (p = 0.0007, $\mu_{\mathrm{DM}} = 10.96$, $\mu_{\mathrm{M}} = 19.75$, $\sigma_{\mathrm{DM}} = 5.67$, $\sigma_{\mathrm{M}} = 9.54$)

The results of the three statistical tests allow us to enforce the positive results, and thus, the positive answers for the three questions defined in the evaluation. Individual values per participant for the BT and ES metrics are available online.[14]

5.5 Threats to Validity

In the evaluation, we assessed that DataMock was significantly more efficient in terms of time and output model quality than manual data model construction that use mockups only as a mere requirement artifact. When analyzing data model quality, in addition to the taken Error Score metrics we also noted that DataMock allowed building more *precise* data models in the sense that unnecessary classes or classes not explicitly required by end-users (simulated by professors) were less common.

We have found, however, a number of potential threats to our experiment's validity. In the following list, we enumerate such threats and explain how we tried to mitigate them:

- **Errors in Control Model**. The Control Model built was an essential artifact to compute the ES metric. Thus, errors found in it can compromise the values obtained for this metric. To mitigate this threat, we had the Control Model built by members of the experiment organization team, all of them with more than 7 years of experience in teaching and applying data modeling in real applications.
- **Modeler's lack of experience**. Lack of data modeling experience by experiment's participants can lead to non-uniform results—for instance, taking too much time to build models with particular features. To mitigate this threat, every participant was subjected to an interview before the experiment to assess they had enough knowledge and practical experience in the field.
- **Modeler's previous experience in similar applications**. Having previous experience in developing or regularly using applications similar to the three involved in the experiment can affect the results since participants can build models faster or more completely than other participants with less knowledge in this context. We mitigated this threat including a set of questions in the interviews to assess that modelers have no experience in such domains.
- **Non-balanced modeler experience**. Modelers with more experience can build models quicker (for instance, quickly identifying patterns that already applied in other applications) based on their previous experiences in comparison with those that have less experience in the field. To mitigate this issue, as we commented before, we assured that they had passed the basic courses in the Computer Science degree that included concepts of data modeling using E-R diagrams and UML Class Diagrams. In addition, we checked that there were no more than 2 years of difference regarding data modeling in the academy or industry between the participants.
- **Application complexity**. The complexity of the different applications chosen can be in favor of DataMock. To mitigate this issue, we chose applications with different complexity (mockup size and class/attribute density, see Table 8), which leaded to the conclusion that while DataMock performed better in applications with more complexity, it also performs better when the complexity

[14] DataMock Stats for BT and ES metrics—https://docs.google.com/spreadsheets/d/1Sv4qLUdI87n23ERMndPSqGZcn4fBqHvfIu4ksUXc_Ts/.

of the applications is lower. It is important to say that, while the complexity of the different applications can vary, mockups provided for all of them included the core functionality that the site must provide. Thus, we are able to affirm that DataMock allows tackling data modeling in an effective and efficient way when the application size and complexity is adequate.

6 Conclusion and Further Work

In this paper we presented DataMock, a methodology that provides an iterative data modeling strategy using mockups and a simple language based on tags. Throughout this paper we described the theoretical, technological, and methodological aspects of the approach. We also described a controlled experiment we conducted to evaluate it. Experiment results affirm that DataMock provides a more productive and traceable method for data modeling than manual modeling, since adopting it leads to better quality and more traceable models in sensibly less time. The approach benefits from the use of mockups, a current trend in the industry with statistically proven advantages [16]. Since end-users can understand both mockup concepts and natural language tags in DataMock, data modeling with DataMock results natural for them. This leads to a better communication between development and end-user teams, which implies less errors and a more productive modeling process. We argue that the main advantages and innovation of DataMock rely on its mockup-centered data modeling approach. In fact, this allows developers to:

1. Get full traceability of data model elements from scratch. This is possible since every data model specification defined through a tag is linked to a mockup element. In addition, thanks to the expressiveness of the tag language, the derived data models need no major modifications—they are generated by just *merging* the content of each individual tag.
2. Detect domain-related inconsistencies, by iteratively analyzing and comparing the specifications distributed among tags.

During our evaluation, we also found interesting functionalities that we can add to the DataMock tagging language, like calculated fields. We are also working on including text-analysis techniques and algorithms to DataMock, in order to infer simple tags automatically by detecting classic widgets and text patterns in mockups, and finally provide more agility to the tagging process.

Through our controlled experiment we also detected that the tagging strategy used presents a number of usability issues. Thus, we are working on an enhanced version that includes several semiautomated tag refactorings—like, for instance, attribute or class renaming—within the same tool. We expect that such refactorings will reduce even more the data modeling time required by DataMock in comparison with manual data modeling approaches. For this reason, we are planning to conduct an extra controlled experiment to assess the impact of introducing such features in the data modeling approach. In this context and with these additions, we are also considering additional evaluations in order to confirm DataMock's advantages in broader environments—e.g., considering different types of applications or larger development teams.

Since the approach relies on tagging mockups as much as possible, tags can take a lot of physical space from the visual point of view, especially if End-user-Oriented Grammar is used. This makes harder for analysts or developers using the approach to identify and understand the tags that have been placed over mockups. On the other hand, and despite the fact that the controlled experiment results were positive regarding the time taken to specify data models, it also threatens the efficiency of the modeling approach. To tackle this potential improvement we are enhancing our tooling to allow to: (1) summarize tags through different strategies like hiding those not applied over `CompositeWidgets` (i.e., let "more granular" tags only), (2) simplify some of the tag constructs (for instance, allowing expressions like `.attribute` instead of `Class.attribute` when some class context is given). These improvements, in addition to increase tag readability, will allow to place tags faster than how it can be done right now. However, a detailed study need to be accomplished to analyze how these simplifications can affect assisted error detection. Finally, another feature that we are planning to introduce is supporting data model visualization through the integration with some API or Web graphical framework like PlantUML.[15]

In addition, DataMock is intended to model applications which are related to only one database and vice versa. While mockups of different applications can help building a unified database for the different applications, currently DataMock does not provide support for federated database contexts—where two or more applications use two or more different databases. Thus, we plan to add optional database or *data source* specifications to the tag set introduced in this paper in order to provide support for such contexts.

As aforementioned, DataMock is not a full-cycle model-driven development approach but an approach that uses formal annotations and (when possible) UI widgets to help building data models in a faster or less error-prone way. However, it can be effectively integrated into existing model-driven approaches if data models generated with our tool can

[15] PlantUML: Open-source tool that uses simple textual descriptions to draw UML models—http://plantuml.com/.

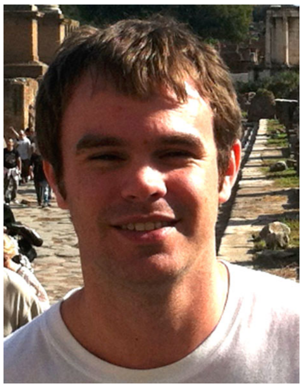
be imported to the tooling provided by such methodologies. Thus, this is another further work path that we are pursuing.

References

1. Topi, H., Ramesh, V.: Human factors research on data modeling. J. Database Manag. **13**(2), 3–19 (2002). doi:10.4018/jdm.2002040101
2. Davies, I., Green, P., Rosemann, M., Indulska, M., Gallo, S.: How do practitioners use conceptual modeling in practice? Data Knowl. Eng. **58**(3), 358–380 (2006). doi:10.1016/j.datak.2005.07.007
3. Pinheiro, F.A., Goguen, J.A.: An object-oriented tool for tracing requirements. IEEE Softw. **13**(2), 52–64 (1996). doi:10.1109/52.506462
4. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In Proceedings of IEEE International Conference on Requirements Engineering, pp. 94–101. (1994) doi:10.1109/ICRE.1994.292398
5. Fliedl, G., Kop, C., Mayr, H.C., et al.: Deriving static and dynamic concepts from software requirements using sophisticated tagging. Data Knowl. Eng. **61**(3), 433–448 (2007). doi:10.1016/j.datak.2006.06.012
6. Kop, C., Fliedl, G., Mayr, H.C.: From Natural Language Requirements to a Conceptual Model. In: Proceedings of the First International Workshop on Evolution Support for Model-Based Development and Testing (EMDT2010), CEUR-WS.org, pp. 67 – 74 (2010)
7. Gorschek, T., Tempero, E., Angelis, L.: On the use of software design models in software development practice: an empirical investigation. J. Syst. Softw. **95**, 176–193 (2014). doi:10.1016/j.jss.2014.03.082
8. Liu, W., Easterbrook, S., Mylopoulos, J.: Rule Based detection of Inconsistency in UML Models, pp. 106–123 (2002)
9. Escalona, M.J., Urbieta, M., Rossi, G., Garcia-Garcia, J.A., Luna, E.R.: Detecting Web requirements conflicts and inconsistencies under a model-based perspective. J. Syst. Softw. **86**(12), 3024–3038 (2013). doi:10.1016/j.jss.2013.05.045
10. Chang, C.K., Zhu, H.: Specifications in software prototyping. J. Syst. Softw. **42**(2), 125–140 (1998). doi:10.1016/S0164-1212(98)10004-3
11. Hussain, Z., Holzinger, A., Slany, W.: Current state of agile user-centered design?: A survey. In: Proceedings of the 5th Symposium of the Workgroup Human–Computer Interaction and Usability Engineering, pp. 416–427. Springer, Berlin (2009)
12. Ferreira, J., Noble, J., Biddle R.: Agile development iterations and UI design. In: AGILE 2007 Conference, IEEE Computer Society: Washington, DC, pp 50–58 (2007)
13. Ton, H.A.: Strategy for balancing business value and story size. In: AGILE 2007 Conference, IEEE Computer Society: Washington, DC, USA, pp. 279–284 (2007)
14. Rivero, J.M., Rossi, G., Grigera, J., Luna, E.R., Navarro, A.: From interface mockups to web application models. In 12th International Conference on Web Information System Engineering, Sydney, Australia, pp. 257–264 (2011)
15. Mukasa, K.S., Kaindl, H.: An integration of requirements and user interface specifications. In: 6th IEEE International Requirements Engineering Conference, pp. 327–328, IEEE Computer Society: Barcelona, Catalunya (2008)
16. Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., Astesiano, E.: On the effectiveness of screen mockups in requirements engineering. In: 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM Press: New York (2010)
17. Rivero, J.M., Grigera, J., Rossi, G., Luna, E.R., Montero, F., Gaedke, M.: Mockup-Driven development: providing agile support for model-driven web engineering. Inform. Softw. Technol. **56**(6), 1–18 (2014). doi:10.1016/j.infsof.2014.01.011
18. Zhang, J., Chung, J.Y.: Mockup-driven fast-prototyping methodology for web application development. Softw. Pract. Exp. **33**(13), 1251–1272 (2003). doi:10.1002/spe.547
19. Forward, A., Badreddin, O., Lethbridge, T.C., Solano, J.: Model-driven rapid prototyping with Umple. Softw. Pract. Exp. **42**(7), 781–797 (2012). doi:10.1002/spe.1155
20. Brambilla, M., Fraternali, P.: Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. Morgan Kaufmann, Burlington (2014)
21. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: Uml-based web engineering: an approach based on standards. In: Rossi, G., Pastor, O., Schwabe, D., Olsina, L. (eds.) Web Engineering: Modelling and Implementing Web Applications. Springer, London (2008)
22. Jwo, J.-S., Cheng, Y.C.: Pseudo software: A mediating instrument for modeling software requirements. J. Syst. Softw. **83**(4), 599–608 (2010). doi:10.1016/j.jss.2009.10.042
23. Ramdoyal, R., Cleve, A.: From pattern-based user interfaces to conceptual schemas and back. In: Proceedings of the 30th International Conference on Conceptual Modeling—ER 2011, Brussels, Belgium, pp. 247–260 (2011)
24. Rosenberg, D., Stephens, M.: Collins–Cope. Agile development with ICONIX process—people, process, and pragmatism. (Apress ed) (2005)
25. Fortuna, M.H., Werner, C.M.L., Borges, M.R.S.: Info Cases: Integrating Use Cases and Domain Models. In: 16th IEEE international requirements engineering conference (RE), vol 0. IEEE Computer Society: Catalunya, Spain, 2008; 81–84. DOI:http://doi.ieeecomputersociety.org/10.1109/RE.2008.43
26. Kulak, D., Guiney, E.: Use cases: requirements in context. Addison-Wesley, Boston (2004)
27. Linehan, M.H.: SBVR Use Cases. Bassiliades N., Governatori G., Paschke A. (eds). Lecture Notes in Computer Science Volume 2008; **5321**: 182–196. doi:10.1007/978-3-540-88808-6
28. Bajwa, I.S., Choudhary, M.A.: From natural language software specifications to UML class models. Zhang, R., Zhang, J., Zhang, Z., Filipe, J., Cordeiro, J. (eds). Lecture Notes in Business Information Processing, vol. 102, pp. 224–237. (2012). doi:10.1007/978-3-642-29958-2
29. Winkler, S., Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. **9**(4), 529–565 (2009). doi:10.1007/s10270-009-0145-0
30. Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, JL.: Consistency Problems in UML-Based Software Development. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A., (eds). Lecture Notes in Computer Science, vol. 3297, pp. 1–12. (2005) doi:10.1007/b106725
31. Usman, M., Nadeem, A., Kim, T., Cho, E.: A survey of consistency checking techniques for UML models. In: Proceedings of Advanced Software Engineering and Its Applications (ASEA) (2008). doi:10.1109/ASEA.2008.40
32. LaRoche, C.S., Traynor, B.: User-centered design (UCD) and technical communication: The inevitable marriage. In: 2010 IEEE international professional communication conference. IEEE, pp. 113–116 (2010). doi:10.1109/IPCC.2010.5529821
33. Nielsen, J.: The usability engineering life cycle. IEEE Comput. **25**(3), 12–22 (1992). doi:10.1109/2.121503
34. Constantine, L.: Canonical Abstract Prototypes for Abstract Visual and Interaction Design. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J., (eds.) Springer, Berlin (2003). doi:10.1007/b13960
35. Martin, A., Biddle, R., Noble, J.: The XP customer role in practice: Three studies. In Agile Development Conference, IEEE Computer Society: Salt Lake City, Utah, USA, pp. 42–54 (2004)

36. Muller, M.J.: The Human–Computer Interaction Handbook. In: Jacko, J.A., Sears, A. (ed.) pp. 1051–1068, L. Erlbaum Associates Inc, Hillsdale (2003)
37. Batra, D.: Conceptual data modeling patterns: representation and validation. In: Wang, J. (ed.) Data Warehousing and Mining: Concepts, Methodologies, Tools, and Applications, pp. 280–302. Hershey Publisher, Hershey (2008)
38. Vidya Sagar, V.B.R., Abirami, S.: Conceptual modeling of natural language functional requirements. J. Syst. Softw. **88**, 25–41 (2014). doi:10.1016/j.jss.2013.08.036
39. Rivero, J.M., Robles Luna, E., Grigera, J., Rossi, G., Improving user involvement through a model-driven requirements approach. In: 2013 International Workshop on Model-Driven Requirements Engineering (MoDRE). Rio de Janeiro, Brazil, pp. 20–29 (2013). doi:10.1109/MoDRE.2013.6597260
40. Wills, L.M., Kordon, F.: Rapid system prototyping. J. Syst. Softw. **70**(3), 225–227 (2004). doi:10.1016/S0164-1212(03)00070-0
41. Ghabi, A., Egyed, A.: Exploiting traceability uncertainty among artifacts and code. J. Syst. Softw. **108**, 178–192 (2015). doi:10.1016/j.jss.2015.06.037
42. Cohn, M.: User stories applied: for agile software development. Addison-Wesley, Boston (2004)
43. Rivero, J.M., Rossi, G., Grigera, J., Burella, J., Robles Luna, E., Gordillo, S.: From mockups to user interface models: an extensible model driven approach. In: Proceedings of the 10th International Conference in Web Engineering (ICWE'10), Springer, Berlin, pp. 13–24 (2010)
44. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M., Trevisan, D.: USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In: Proceedings of the ACM AVI 2004 Workshop
45. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley, Boston (1999)
46. Beijering, K., Gooskens, C., Heeringa, W.: Predicting intelligibility and perceived linguistic distance by means of the Levenshtein algorithm. Linguist. Neth. **24**, 13–24 (2008)
47. Basili, V., Caldiera, G., Rombach, D.: The Goal Question Metric Approach. (1994)
48. Nelson, H.J., Poels, G., Genero, M., Piattini, M.: A conceptual modeling quality framework. Softw. Qual. J. **20**(1), 201–228 (2011). doi:10.1007/s11219-011-9136-9
49. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin (2012)
50. Cliff, N.: Dominance statistics: ordinal analyses to answer ordinal questions. Psychol. Bull. **114**(3), 494 (1993)

José Matías Rivero received a PhD in Computer Science from the National University of La Plata (UNLP), Argentina, in 2015. Currently, he is a postdoc scholar at the National Scientific and Technical Research Council (CONICET), the most important public research institute in that country. His research interests lie in the area of Web Engineering and more specifically include the topics of UI Prototyping, Mockups, User-Centered Design, and User-Centered Requirements Engineering. He has co-authored more than 15 papers published in international conferences and journals.



Julián Grigera is a PhD student and teaching assistant at Facultad de Informática, Universidad Nacional de La Plata, Argentina, where he is a member of the Research and Development in Advanced IT Lab (LIFIA). His research interests are Web Engineering and Web Usability.

Damiano Distante is Associate Professor of Computer Science at the Unitelma Sapienza University, Italy. He holds a PhD in Information Engineering and a Master Degree in Computer Science and Engineering from the University of Salento, Italy. His main field of research is software engineering in general and software evolution and web engineering in particular. His research interests include: design and model-driven development of Web applications, evolution of Web systems, e-learning methodologies and technologies, data mining and information retrieval techniques. He co-authored more than 50 papers in referred international journals and proceedings of international conferences. He is member of the IEEE Computer Society.

Francisco Montero is Associate Professor of Computer Science at the University of Castilla-La Mancha, Spain. During his master degree studies at the University of Castilla-La Mancha and the University Polytechnic of Valencia (Spain), he was a holder of several research scholarships funded by the Regional Government of Castilla-La Mancha and the National Government. He got his bachelor degree and PhD in the University of Castilla-La Mancha. He is currently an active collaborator of the LoUISE Research group of the University of Castilla-La Mancha. His current research interests are usability, accessibility, User Interfaces, and human–computer interaction.

Gustavo Rossi is Full Professor of Web Engineering at Facultad de Informatica, Universidad Nacional de La Plata, Argentina, and researcher at Conicet, Argentina. He is director of LIFIA (Research Lab on advanced informatics) at UNLP, and he is a member of IEEE and ACM. He has published more than a hundred papers on top conferences and journals. His research interests include model-driven web engineering, requirements engineering, and human–computer interaction.