Open Access

Open Journal of Information Systems (OJIS)
Volume 2, Issue 2, 2015

http://www.ronpub.com/journals/ojis
ISSN 2198-9281

Achieving Business Practicability of Model-Driven Cross-Platform Apps

Tim A. Majchrzak [A], Jan Ernsting [B] and Herbert Kuchen [B]

[A] University of Agder, Gimlemoen 25, 4630 Kristiansand, Norway
[B] ERCIS, University of Münster, Leonardo-Campus 3, 48149 Münster, Germany
{tima, jan.ernsting, kuchen}@ercis.de

Abstract

Due to the incompatibility of mobile device platforms such as Android and iOS, apps have to be developed separately for each target platform. Cross-platform development approaches based on Web technology have significantly improved over the last years. However, since they do not lead to native apps, these frameworks are not feasible for all kinds of business apps. Moreover, the way apps are developed is cumbersome. Advanced cross-platform approaches such as MD^2, which is based on model-driven development (MDSD) techniques, are a much more powerful yet less mature choice. We discuss business implications of MDSD for apps and introduce MD^2 as our proposed solution to fulfill typical requirements. Moreover, we highlight a business-oriented enhancement that further increases MD^2's business practicability. We generalize our findings and sketch the path towards more versatile MDSD of apps.

Type of Paper and Keywords

Regular research paper: *Cross-platform, MDSD, app, business app, mobile, MD^2*

1 Introduction

Businesses increasingly embrace mobile computing. Applications for mobile devices (*apps*) such as smartphones and tablets are not only developed for sale or to directly earn money (e.g. by placing advertisements in them). Rather, enterprises have identified usage scenarios in internal utilization by employees, field service, sales, and customer relationship management (CRM) [37]. Besides some other topics such as security [13] and testing [48], cross-platform development is a major concern when implementing apps[23].

The need for cross-platform development approaches arises from the incompatibility of today's platforms for mobile devices. With Apple's iOS, Google's Android, Microsoft's Windows Phone, and RIM's Blackberry there are at least four major platforms (cf. [20]) that need to be supported in order to reach *most* potential users of an app. Each platform has an ecosystem of its own and differs with regard to development language, libraries, and usage of device-specific hardware – to name just a few factors. Developing separately for each platform currently is the choice. This is an error-prone and extremely inefficient procedure that becomes particularly frustrating when updating existing apps.

Based on the proliferation of adequate frameworks [28], mobile Webapps have become very popular. They are based on web technology, executed by a browser, and hence usable on essentially every platform. Cross-platform approaches based on Web technology such as Apache Cordova [5] (a.k.a. PhoneGap [43]) are suitable for many app projects. They are rather easy to learn, offer good community support and rich literature, and – most notably – can be deployed as *real* apps. This also allows

offering them in app stores and to virtually use them in any way native apps could be used. However, apps based on Web technology are not feasible in all cases [10].

When working with Webapps, their origin in Web technology cannot be neglected. This has been called an *uncanny valley* [17]: a typical Webapps' look & feel is almost "real" (i.e. native) but the app is slightly less responsive. Moreover, even with HTML5 [29] not all device-specific features are supported. Connecting to server backends, as required for most *business apps* [27] (cf. Section 2), is cumbersome and usually inefficient. Finally, apps are developed with a very low level of abstraction. Domain-specific knowledge has to be communicated to developers instead of being built directly into an app; existing models e.g. of business processes or information systems cannot be used even if they would be applicable to the scenario that an app is intended for.

To close the above sketched gap, we have compiled requirements for typical apps used by businesses for purposes different to sales of these apps. To enable effective cross-platform development of business apps, our group has developed a prototype for model-driven development (MDSD [49]) of apps [24]. MD^2 [25] is suitable for some scenarios already but requires further enhancements. Using MDSD for apps differs greatly from current state-of-the-art app development and is less mature. It, however, offers great opportunities despite requiring refinements.

Our article makes a number of contributions. Firstly, it compiles requirements for typical business apps, which have been validated by practitioners. Secondly, it introduces an approach for model-based cross-platform app development that is based on these requirements. And thirdly, it discusses the path towards business practicability of sophisticated cross-platform development frameworks.

This article is structured as follows. Section 2 sketches the background of our work including a discussion of what is a *business app*. In Section 3, our approach to develop cross-platform apps – MD^2 – is introduced. Section 4 characterizes the particularities of a business-oriented enhancement for MD^2. In Section 5 we discuss and generalize our findings before drawing a conclusion in Section 6.

2 BACKGROUND

In the following, we first characterize business apps and propose a set of requirements for them. Next, we introduce model-driven development in general as well as cross-platform approaches for apps. Finally, we discuss related work.

2.1 Characterization of Business Apps

The first step to define requirements is to define what a *business app* is. In our understanding, a business app is used by enterprises for other means than to directly make revenue with it. Consequently, a business app might be used to

a) support business processes,

b) improve the work with suppliers and collaborators, or

c) offer value to customers and thereby become an instrument of customer relationship management or marketing.

To get a first idea of typical business apps, we rely on a study conducted with regional companies [27, 37] augmented with knowledge from working with several partners from industry. Business apps have two constituting and one supporting characteristic. Firstly, business apps are *form-based*. This means that they are made up of standardized elements (so called widgets [40]) such as text fields, check boxes, drop-down fields, and buttons. These are typically aligned on composites that can be nested in each other and placed on windows, tabs, and similar aggregating elements. Thereby, form-based apps resemble text-based forms with the means of graphical user interfaces. Forms typically can be built both with native software development kits (SDKs) and with Web technology. While forms are not limited to simple elements but could e.g. contain *mashups* [58], graphical user interfaces not created with widgets are a rare occurrence. While a game that uses real-time rendered graphics might be used for advertisement purposes, it would not typically be considered to be a business app.

Secondly, business apps are *data-driven*. The three above sketched scenarios require some form of data input, transformation, and output. Admittedly, all but the simplest programs match this definition. However, business apps are used to provide specific information in alignment with their intended purpose, and most of them accept data to select the kind of information provided or to feed a server backend. To make an example, think again of a game. It accepts input and shows the results (such as a moving avatar) but its operation is not driven by data in the same way as, say, an app to look up departure times of trains.

Thirdly, not all but most business apps are linked to one or several *backend* systems. Many apps are fully functional without an active connection to the Internet. They either have no sophisticated business logic at all, or the logic is incorporated with the app. While this might be desirable in many cases, typical business apps rely on a server backend. There are good reasons for this. Companies that develop apps commonly embed these apps

in the (enterprise) information systems landscape. Apps ought to fulfill services in alignment with the services of the company. They can draw from the rich functionality of backends and orchestrate services to their needs. Merely developing an app as a frontend to such services shortens development time. This approach is not only convenient but also takes security concerns into account, since no critical logic is put into apps that are run on inherently insecure [32] devices. Think of an app to support business processes, say a tariff calculator supporting insurance field staff. The way premiums are calculated might be changed at times, but the data required to calculate them basically stays the same. Hence, apps should be data-driven but request premiums from the insurer's backend systems rather than calculating them. In addition, how tariffs are calculated can be seen as a trade secret of an insurer. Thus, keeping them in the secured company backend and only requesting premiums via an additionally secured, standardized Web service is reasonable.

2.2 Requirements of Business Apps

The three properties introduced above characterize business apps. We have identified additional requirements.

In general, it is desirable to develop apps with a high level of abstraction. Business apps are built based on business knowledge. If possible a programmer should describe *what* the app should do and not necessarily *how*, as with object-oriented programming languages such as Java and Objective-C. Moreover, it should be possible to develop an app once and deploy it to any desirable mobile platform.

Since business apps are typically data-oriented, it is reasonable to enable the easy definition of entity types and to provide corresponding create, read, update, and delete (CRUD) operations automatically. Moreover, it is helpful to connect certain input fields of the user interface (UI) to certain local or remote entities (possibly stored in a database on a server) via so-called *data bindings* (cf. [25]). Changes to such input fields will then be automatically propagated to the entity and vice versa. In addition, there should be easy to use and at best declarative ways to ensure that certain input fields are *validated*, i.e. illegal input values are rejected. Such validators can for instance ensure that a German zip code consists of exactly five digits and that letters and special symbols are forbidden. Checking is possible locally or remotely on a server.

For the user interface (UI), all typical widgets including high-level composites such as tabs have to be supported. The UI further needs to support common input methods of mobile devices including gestures such as swiping.

Moreover, there needs to be control of the sequence of showing UI elements, i.e. workflow control. The app needs to be reactive to events such as incoming messages and state changes such as tilting the device (*orientation change*). Finally, business apps increasingly make use of device-specific hardware such as GPS receivers, which should hence be supported.

Business apps also have some non-functional requirements. Often, companies desire a native look & feel [37], which rules out Web applications. The UI needs to be user friendly. Intuitive usage without explanation should be possible. Additionally, the performance of apps needs to be at least acceptable and apps need to be robust in cases of improper use. Considering network conditions, apps should even be resilient, i.e. not only tolerate connectivity problems but resume operation after a connection is restored. Finally, apps should be secure, i.e. it should not be possible to use them in a way that could be harmful to its users or to the enterprise that they have been designed for. An open interface should allow for extensions in case of new sensors and device gadgets becoming standardized.

In general, requirements are neutral with regard to technology. However, not all of them can be fulfilled with an arbitrary technology stack, in particular not solely with HTML5 [45]. Specifically, some requirements can – at least so far – only be addressed with native apps or development methods yielding native source code. Examples are apps that require background processing [45] and advanced security concepts [55]. Moreover, reaching a high-level of abstraction is impossible with most current approaches (cf. Sections 2.4 and 2.5).

When working with the above sketched requirements, it has to be kept in mind that they are generic. Specific business apps might have stricter requirements (besides the functional requirements for the very app) *or* be more open. Nevertheless, our set of requirements can be used as a reasonable starting point.

2.3 Model-Driven Software Development

The idea of model-driven software development (MDSD) [49] is that software and other artifacts such as configuration files or database schemas are not written by hand but generated from a model. A typical way of implementing a corresponding generator is to use a template processor such as Xtend [14] and to fill gaps in a template consisting of source code fragments with details taken from the considered model. In order to support n platforms, n different templates are needed. The model, however, does not have to be changed and can be used for all destination platforms.

A model can be formulated in a classical modelling language such as UML [47]. For a specific area such as business apps for mobile devices, more concise models can often be obtained by using a dedicated *domain-specific language* (DSL) [49].

A framework such as Xtext [57] can be used to specify

the syntax of a DSL in an enhanced Extended Backus-Naur Form (EBNF) and to use the built-in parser generator to create a parser transforming a textual model written in that DSL into an abstract syntax tree, which can then be transformed by the mentioned generator. Xtext is part of the Eclipse Modeling Framework (EMF) [15].

2.4 Cross-Platform Apps

There are several ways for developing apps for multiple platforms. The currently available approaches are summarized in Figure 1. The figure also shows a hierarchy derived from the similarities of the approaches.

In general, *cross-platform* and *native development* can be distinguished. Native development uses the platform-specific software development kits (SDKs) along with the applicable development tools. The programming language is bound to the platform; for example, Java is used for Android apps whereas Apple requires usage of Objective C (or, newly, Swift [30]). Consequently, development is carried out for each platform separately. Development effort increases almost linearly with each additional platform – of the typical activities carried out in software engineering, only requirements engineering is more or less shared while design, implementation and testing are carried out independently on each platform. In sharp contrast, *cross-platform approaches* follow an "implement once – run everywhere" principle (at least in theory).

Cross-platform apps can be developed following one of two main paradigms. A *runtime environment* abstracts from the native interface of a platform. *Generative approaches* allow to develop an app once and then to generate native source code for each supported platform.

Generative approaches can be realized in two possible ways. Firstly, *model-driven software development* (MDSD) can be employed. We follow this approach (see Section 3): an app is described using a (typically textual or graphical) modeling language, which is independent of an actual platform. Such a model is transformed to native, platform-specific code. Transformation is usually done by using tools. Secondly, *transpiling* translates the single used programming language to platform-specific ones. Examples for both approaches including their current shortcomings are discussed in [23].

Cross-platform approaches based on runtime environments use Web technologies such as HTML5, Cascading Style Sheets (CSS), and JavaScript for develpping apps. *Mobile Webapps* are the simplest approach: an app is solely built with Web technologies. It therefore can be used on any device offering a Web browser but is limited to elements known from Web sites. Moreover, deployment to app stores typically is not possible. *Hybrid* approaches go one step further by providing native packaging. Apps still are Webapps by character but can be deployed to native platforms. Despite the still Web-like look & feel, hybrid approaches have become very popular due to the ease of development. Finally, a *self-contained runtime* allows development based on Web technology but offers a bridge to native GUI elements. Thereby, apps become closer to a native look & feel at the price of a performance penalty.

For an extended introduction including an evaluation of approaches and examples please refer to [23] and [41]. Web development vs. native development is discussed in detail by Charland and Leroux [10].

2.5 Existing Approaches

Cross-platform app development approaches have been mentioned as early as 2009. Miravet et al. present their framework DIMAG, which is based on State Chart eXtensible Markup Language (SCXML) [38]. Even when neglecting the different concept, their approach is hardly comparable to ours due to the missing focus on business apps. In terms of app development, 2009 is *long* ago and, thus, native development versus Web development was not yet a topic of interest, either. Nevertheless, the approach is notable for the sketched ideas.

As outlined in Subsection 2.4, cross-platform approaches can take different forms. Even what is considered a cross-platform approach might vary – practitioners sometimes employ looser definitions (cf. the comparison by [11]). Besides HTML5 [29], Apache Cordova [5] is a framework that utilizes Web technology but also supports accessing native device features, i.e. allowing the development of *hybrids*. Yet, its Web foundations negatively impact aspects such as app responsiveness. Other approaches build upon a *self-contained runtime* such as Titanium [6]. These typically operate on a custom scripting language that are interpreted by runtime environments.

Cabana [12], AXIOM [33], applause [7], and Xmob [34] are representatives of *generative approaches*. Cabana focuses on development of apps in the context of higher education. It does so by providing a GUI to manipulate graphic models of app representations. Furthermore, it allows to implement customized code, if need be. However, interactions with backends are neglected and using other platforms for achieving this is advised (cf. [12, p. 533f.]). Cabana apparently has been discontinued [50]. AXIOM takes a technical stance as it features aspects of UML and uses the programming language Groovy [21]. Moreover, it does not fully automate intermediate steps of code generation [25]. applause and Xmob are most similar to MD^2: they provide DSLs, too. Yet, applause is mostly restricted to displaying information and does not provide a DSL tailored to describing business apps. Xmob's DSL features aspects alike those of MD^2, but does not provide the needed tools up to now.

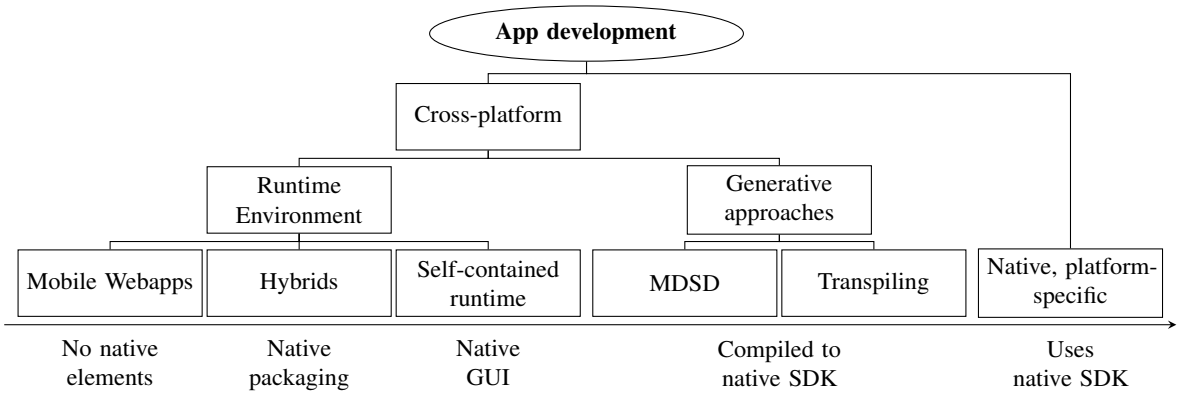


Figure 1: Categorization of Cross-Platform Approaches (adapted from [27, p. 63])

MDSD for app development is not only employed in cross-platform contexts. Franzago, Muccini, and Malavolta present work on a modelling framework for data-intensive apps [19]. Recent advancements like this could go along MD^2's development and lead to future improvements. Even though targeting the Web, approaches such as WebDSL [3] and mobl [2] provide domain-specific languages for Web development, in the latter case focusing on HTML5 and app development. There is no direct connection to our work, though.

Some interesting approaches come from industry. Among others, IBM [1] and SAP [54] are working on cross-platform technologies to complement their products. Specialised companies such as Mendix [22] and WebRatio [4] also contribute to this movement. The latter explicitly focusses on model-driven technologies.

3 Cross-Platform Development with MD^2

In this section, we first motivate our approach with a real-world example. We then successively introduce its architecture, features, and domain specific language. Finally, we name current limitations.

3.1 Introductory Example

A corporation has a customer relationship management (CRM) system. It wants to provide access to its sales staff, allowing them to record prospective customers as part of their acquisition process through their mobile devices. However, the CRM does not support mobile devices but offers an application programming interface (API) for third party applications. In addition, the corporation has no prior knowledge of mobile app development and wants to use MD^2 to integrate its CRM and the mobile apps that are to be created. The scope of the example[1] is limited to keep it brief: it will exclusively focus on recording prospective customers through mobile apps.

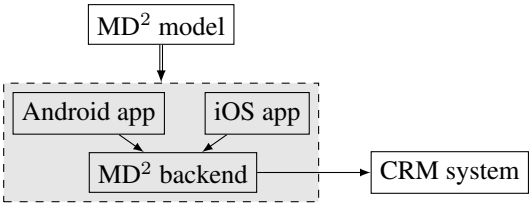


Figure 2: Basic Architecture

3.2 Architecture and Features of MD^2

Using the textual MD^2-DSL the corporation defines a model. From that MD^2 model the artifacts in the shaded area of Figure 2 are generated. In fact, these generated artifacts represent executable code for the mobile apps as well as the backend and expose the following properties:

- Mobile apps are automatically linked to the backend.
- The generated MD^2 backend already constitutes a fully Java Enterprise Edition (JEE) compliant application container including an entity model that can be persisted through Java's Persistence API (JPA).
- Developers only implement the "glue code" in the generated MD^2 backend to link it to the corporation's CRM API (corresponds to the link from MD^2 to the CRM in Figure 2). This typically is a straightforward task.

[1] Underlying sources can be obtained from `http://git.io/md2-multi-valued-elements` and `http://git.io/md2-testApps-crmContactManager`.

3.3 MD^2-DSL

A MD^2-DSL model is structured according to the well-known *model-view-controller* pattern [9]. Thus, it consists of three parts specifying the model, view, and controller component of an app. In our example, the model (see Listing 1) defines just a single entity type `Contact` (with first name, surname, etc.) and an enumeration type `AcquisitionState`.

Listing 1: MD^2 model

```
entity CONTACT {
  firstname : string
  surname : string
  phone : integer (optional)
  email : string (optional)
  state : ACQUISITIONSTATE
}

enum ACQUISITIONSTATE {
  "Prospective", "Acquiring", "Acquired", "Rejected"
}
```

As depicted in Listing 2, the corresponding view first fixes a layout. Here, `FlowLayout` has been chosen, which displays the different widgets from top to bottom and from left to right on the screen. The view component `addContactView` displays a contact and a button `Add`. Here, the layout of a contact is automatically inferred from the structure of the results provided by a *content provider*, namely `contactContentProvider`. This content provider is defined in the controller part of the model (see Listing 3 – package definitions in this and all following listings have been stripped for brevity). As can be guessed from its name, this content provider will provide a contact. Thus, text fields for first name, surname, and so on as well as corresponding labels will be displayed. The semantics of the button `Add` will be determined in the controller component explained below.

Listing 2: MD^2 view definition

```
FlowLayoutPane ADDCONTACTVIEW (vertical) {
  AutoGenerator autoGenerator {
    contentProvider contactContentProvider
  }
  Button addButton ("Add")
}
```

The corresponding controller component (see Listing 3) first specifies some meta-information (such as `appVersion` and `modelVersion`). Moreover, it determines the initial view component of the app (here: `addContactView`) and the action, which should be executed when the app is started (here: `startUpAction`). Then, it defines *content providers* and *actions*, which are executed when certain events are observed. A content provider can be e.g. located on a server. In our example, the content provider `contactContentProvider` is located on the server with URI `http://md2.crm.corp.com/`. Moreover, the initial `startUpAction` binds `addAction` to the `onTouch` event of the button `addButton` occurring in the `addContactView` described above. Thus, if this button on the screen is touched, the `addAction` will be executed. As shown in Listing 3, the `addAction` will cause the content of the text fields corresponding to the displayed contact to be stored by the content provider `contactContentProvider`. Thus, the inputs will be synchronized with the contents of the corresponding contact entity.

Listing 3: MD^2 controller definition

```
main {
  appName "CRM contact manager"
  appVersion "1.0"
  modelVersion "1.0"
  startView ADDCONTACTVIEW
  onInitialized startUpAction
}

contentProvider CONTACT contactContentProvider {
  providerType crmSystem
}

remoteConnection crmSystem {
  uri "http://md2.crm.corp.com/"
}

action CustomAction startUpAction {
  bind action addAction on ADDCONTACTVIEW.addButton.onTouch
}

action CustomAction addAction {
  call DataAction (save contactContentProvider)
}
```

3.4 Underlying Technology

MD^2-DSL is based on the Xtext language framework [57] and is described in detail in [26]. MD^2 utilizes Xtend [14] for processing its models as well as generating code for all target platforms.

Code generators transform the textual model to compilable code. These outputs are embedded into preconfigured projects, that are generated at the same time, and

can be compiled using the specific target platform's tools, i.e. Eclipse for the JEE backend as well as the Android app and Xcode for the iOS app.

As the generated apps communicate with the backend through RESTful web services [46] exchanging JSON messages, the backend can be substituted with arbitrary web services that exhibit the same properties. Thus, it is possible to integrate our approach with backends that are for example based on LAMP [35] or .NET technology stacks. Additionally, MD^2 can be extended to generate according backend code, if need be.

3.5 Current Limitations

MD^2 is an academic prototype despite the cooperation with practitioners and its application in first practical projects. Thus, it poses some limitations. Some of these are inherent to the approach of using MDSD. Most, however, can be considered as work-in-progress boundaries that can be overcome in the future.

Firstly, a detailed evaluation of MD^2 has not yet been done. Our approach has been refined and internally evaluated several times but no field study, detailed analysis, or competitive analysis in comparison to other approaches in a commercial context has been conducted.

MD^2 is no "one-size-fits-all" approach. MDSD for apps most likely will never be suitable in some scenarios (e.g. apps that render their graphics on-the-fly). While we have business apps in mind and theoretically fixed this scope, refinement and acknowledgement of the very area to use MD^2 remains a future challenge. Nevertheless, you could consider MD^2 as "one-DSL-fits-most-cases".

From a technological viewpoint, some additions to MD^2 are interesting. In particular, it currently is not possible to add custom code. While this might be undesirable on the frontend (i.e. the app – the MDSD approach would become blurred), it would be helpful to have better support for custom additions to a backend. It is designed as a blueprint already but extension beyond simple connections requires much manual work. Possibly suitable approaches discussed in the literature are the generation gap pattern [18, pp. 571ff.][52], protected regions [49, p. 29][31], and dependency injection [44][16]. The latter appears to be the most elegant solution. However, a discussion quickly becomes very technical (and little business-related) and, thus, is out of scope of this article.

There are some minor limitations. In fact, they are not limitations in a strict sense but topics not yet dealt with. As already argued, testing of apps is cumbersome. Testing (and *checking*) MD^2 should be significantly easier and at the same time very powerful since a model can be used as the basis of testing (cf. [8, 51]). Nevertheless, we have not yet addressed testing explicitly. Moreover, energy efficiency has not been considered – it is an increasingly important yet often neglected topic [39, 42]. Finally, a business-oriented framework might require to support mobile device management (MDM). MD^2 does not provide an interface to MDM – as to our knowledge no framework currently does.

As a final remark, there are no specific security features built into MD^2. On the one hand, there is no need for much security. Due to the DSL, it is hardly possible to use MD^2 maliciously anyway. Much more important, however, is the fact that business logic typically resides (almost exclusively) on the backend. Therefore, proper authorization and authentication is required – both are possible with the means that MD^2 offers. Security of the business logic and the database(s) used is a task of backend operations, which is extremely helpful. On the other hand, with an extended evaluation by businesses, scenarios might arise that require additional security features that we did not yet consider. However, extending MD^2 in such cases should be hassle-free.

4 Business-oriented Enhancement

While the core of MD^2 has been described above, our framework offers additional features. In the following, support of *multi-valued elements* is described with special focus on their business-orientation, i.e. the match with the requirements described in Subsection 2.2.

When considering relationships between entities, two maximum cardinalities come to mind: single and multiple. Relationships with at most a single entity on the referenced side can already be defined through MD^2 models (e.g. one customer → one address). This did, so far, not hold true for relationships with multiple entities on the referenced side (e.g. one customer → many addresses). At first, multi-valued elements in MD^2 were implemented only to a certain degree. In fact, they were supported by the content providers but not on the view or controller level. Our recent work on MD^2 tackled this shortcoming and refined it to provide support for multi-valued elements.

Listing 4: Augmented MD^2 model

```
1  entity CONTACT {
2    ...
3    interactions : INTERACTION[]
4  }
5
6  entity INTERACTION {
7    interactiondate : date
8    occasion : string
9    ...
10 }
```

Regarding our previous example, sales representatives need to get access to customer records as well as pre-

vious interactions. For that, the entity type `Contact` in Listing 4 is augmented with a list of interactions as denoted by the array-like syntax. In addition, the model is extended with a new entity type `Interaction` (with interaction date, occasion, etc.).

To display customer details and interactions, view and controller definitions require changes, too. Within the `contactDetailView` a `List` element is used to define a list view of customer interactions (see Listing 5). As defined by the `itemtext` value, the list view displays the occasion for each associated customer interaction. The controller definition is omitted here but is augmented to bind actions and data accordingly. Alternatively, the view definition of lists can be automatically generated by the `AutoGenerator` element (cf. Listing 2).

Listing 5: Augmented MD^2 view definition

```
1 FlowLayoutPane CONTACTDETAILVIEW ( ↩
      vertical) {
2 ...
3   List interactionsList {
4     itemtype INTERACTION
5     itemtext INTERACTION.^occasion
6     listtype plain
7   }
8 }
```

Summing up, multi-valued elements might be omitted at first but are needed to address functional requirements typically found in business apps. We have made a suggestion how to cope with multi-valued elements in domain-specific languages for app development such as the one of MD^2.

As an interesting remark, the implementation of multi-valued elements in the generators for Android and iOS showed significant differences. While details are not within the scope of this article, it is a good example for differences between the platforms that approaches, which provide native code, must overcome. At the same time, developers are relieved from understanding how (and why) similar concepts are treated different on distinct platforms.

5 DISCUSSION

The insights presented so far are heterogeneous on the one hand, interconnected on the other hand. While the requirements for increased usage of apps by enterprises are clear, app development – in particular across platforms – is no trivial task. With MD^2 we have proposed a solution to address the requirements of business apps by using MDSD. The path towards business practicability of sophisticated cross-platform development is still steep (but rewarding), though, as will be discussed in the following.

5.1 The Path towards Business Practicability

After describing requirements of business apps, introducing our approach including business-oriented details, and discussing its merits and limitations, one question remains: how could the path towards business practicability of model-driven techniques for app development look like? In particular, what will make approaches such as MD^2 feasible beyond its academic, prototypic state and applicable for commercial use?

There is little doubt in the general feasibility of the approach. As discussed earlier, MDSD allows to perfectly address requirements of business apps. Despite differences between academia and industry [56], MDSD is nowadays also embraced by practitioners. MD^2 has been developed in cooperation with practitioners and even tested by them in real-world projects. However, several factors hinder its wider usage, and in parts wide usage of MDSD for app development in general:

- *Scale*: MD^2 is hardly known by businesses. New approaches, particularly if they propose a paradigmatic shift, require a baseline of users to prosper. The dynamics of achieving a critical mass of users (as currently few approaches have) are complex.
- *Performance*: Even though the perceived performance is fine, benchmarking of MD^2 apps and comparing performance figures to that of natively developed apps would increase the trust in MD^2's feasibility.
- *Long-term support*: Approaches require the promise to be supported for at least a few years. Otherwise the risk of using them in projects is too high for corporate users. MD^2 is open source, which is a first step, but a community around an approach needs to be established.
- *User-friendliness*: we have not yet assessed how easy it is to work with MD^2. Praises of technological superiority are meaningless if other approaches win the race due to their ease of use. Without the aim to criticize Apache Cordova [5], there is little question that it has inherent shortcomings. However, it is easy to use and yields *good* results. Therefore, its cost-effectiveness is high – its wide adoption is, thus, unsurprising and justified.
- *User feedback*: we have already refined MD^2 but many more evaluation and refinement iterations are required.
- *Extension*: Currently, it is possible to reach *some* boundary of MD^2 quite quickly. This particularly applies to the number of supported platforms. Extending it to platforms such as Windows Phone or

Blackberry is laborious and time-consuming, yet not very challenging (but for platform-specific particularities).[2] Extended support remains an important task of future work because it would make the approach much more attractive for businesses.

- *Competition*: Looking at related work (Section 2.5), Web technology-based approaches are predominant. Another convincing MDSD app development tool would stimulate discussion and foster the path towards better MDSD support in the field.

We deem particularly the last point to be very important. It cannot be expected that MDSD for app development will see a land-rush and impose a revolution for app development. It is one idea among others for the much-desired cross-platform capability of modern app development approaches. However, we believe that MDSD can facilitate *better* development practices by being refined evolutionary and by contribution to the discussion about software engineering practices for apps. Thus, experiences from works such as ours can contribute to other cross-platform approaches *not* using MDSD as well.

Without doubt, further enhancements of MD^2 are needed. These enhancement typically also foster contributions to theory. Despite following a road towards business practicability, MD^2 remains a prototype and we strive for discussing advancements with the scientific community. Thus, evaluation of enhancements will allow improvements beyond MD^2 – ideally, even beyond MDSD for app development.

Finally, is has to be found out for which types of apps MDSD is feasible. While MD^2 is an approach for business apps, MDSD is not conceptually limited to this domain. Due to the nature of domain-specific languages, it makes sense to set conceptual boundaries for development frameworks. Nevertheless, while MD^2 will never be the choice to develop games, there are no reasons to believe that a model-based language for gaming apps could not be feasible. However, which domains of app development should be (and will be) addressed by MDSD in the future is very hard to forecast. E-Health is a candidate [53], since form-based layouts and linking with backend systems are typically requirements of healthcare apps. Additionally, other design paradigms could be integrated. For example, white labelling of apps (i.e. developing an app for corporate customers that is deployed to end-users in corporate-specific styles) might be possible in a model-based fashion and combined with MD^2.

[2] While a high number of supported platforms is an important argument for practitioners, it will hardly earn scientific appreciation.

5.2 Future Work

MD^2-DSL was developed using a prototype based approach (from reference prototypes to a DSL). Beginning with a proof of concept, some design decisions regarding the language were not carried out in a consistent fashion. For example, we observed varying levels of abstraction regarding UI widgets. Considering the development of a DSL not as a serial but as a continuous process, these variations are to be aligned to offer more consistent DSL semantics. Thus, the DSL is a main concern of future work.

Due to the complex nature of MDSD, extensive testing of MD^2's components (pre-processors, generators, etc.) was neglected so far. On a more user-centric level, testing of MD^2 apps could be relevant for companies but also for a broader non-MD^2 related audience, too. Improving testability for MD^2 also allows providing stable artifacts (i.e. development tools, plug-ins, etc.) and thus improving accessibility of the framework for novices. Given a test suite, reproducible builds of the artifacts would further improve accessibility to MD^2. As a consequence, testing is the second topic that we will address.

Despite generating native code for the two most common platforms, industry partners expressed interest in generating code for other platforms, be it their own or another one (also cf. with the preceding section). To support custom generators, MD^2 needs to be modified in certain aspects. These modifications and the provision of additional generators are the third topic of future work.

6 Conclusion

We have presented work on the cross-platform development of business apps. In this article, we first described requirements for business apps. They typically are form-based, data-driven, and in most cases have a connection to a company backend. The main non-functional requirement is a native look & feel, which should go along with user friendliness. We then introduced MD^2, an approach for model-driven cross-platform development. Keeping in mind the business implications of our work, we discussed which path towards business practicability of MDSD has to be taken.

Despite the merits of MDSD in app development, it is impossible to forecast whether it will become a dominating technology and the base of future app development. There is plenty of future work. Mobile computing and app development in particular will remain a challenging yet very exiting field of research.

Acknowledgments

We would like to thank Andreas Doepker for his work on the business-oriented enhancements of MD^2. Addi-

tionally, we would like to thank Henning Heitkötter. The figure in Section 2.4 has been inspired by the one he developed for a tutorial at WEBIST 2013, which he jointly held with the first author of this article.

A much shorter and demonstration-oriented version of this article [36] has been presented at the CAiSE 2015 forum in Stockholm.

REFERENCES

[1] "IBM MobileFirst Platform Foundation," 2015, http://www-03.ibm.com/software/products/en/mobilefirstfoundation.

[2] "mobl," 2015, http://www.mobl-lang.org/.

[3] "WebDSL," 2015, http://webdsl.org.

[4] "Webratio," 2015, www.webratio.com.

[5] "Apache Cordova," 2015, http://cordova.apache.org/.

[6] "Appcelerator," 2015, http://www.appcelerator.com/.

[7] "applause," 2014, https://github.com/applause/.

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*. New York, NY, USA: Wiley, 1996.

[10] A. Charland and B. Leroux, "Mobile application development: web vs. native," *Commun. ACM*, vol. 54, pp. 49–53, 2011.

[11] J. Cowart, "Pros and cons of the top 5 cross-platform tools," http://www.developereconomics.com/pros-cons-top-5-cross-platform-tools/.

[12] P. E. Dickson, "Cabana: a cross-platform mobile development system," in *Proc. 43rd SIGCSE*. ACM, 2012, pp. 529–534.

[13] S. M. Dye and K. Scarfone, "A standard for developing secure mobile applications," *Comput. Stand. Interfaces*, vol. 36, no. 3, pp. 524–530, Mar. 2014.

[14] S. Efftinge, "Official Xtend homepage," 2015, http://www.eclipse.org/xtend.

[15] S. Efftinge, J. Köhnlein, and P. Friese, "Build your own textual DSL with tools from the Eclipse Modeling Project," http://eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html.

[16] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004, http://martinfowler.com/articles/injection.html.

[17] M. Fowler, "CrossPlatformMobile," 2011, http://martinfowler.com/bliki/CrossPlatformMobile.html.

[18] M. Fowler, *Domain-Specific Languages*. Addison-Wesley Pearson Education, 2011.

[19] M. Franzago, H. Muccini, and I. Malavolta, "Towards a collaborative framework for the design and development of data-intensive mobile applications," in *MOBILESoft*. New York, NY, USA: ACM, 2014, pp. 58–61.

[20] "Gartner Press Release," 2012. [Online]. Available: http://www.gartner.com/it/page.jsp?id=1924314

[21] "Groovy," 2014, http://groovy.codehaus.org/.

[22] E. Hadley, "New Mendix release enables no-code delivery of multi-device mobile apps for the enterprise," 2014, http://www.mendix.com/press/no-code-enterprise-mobile-app-development/.

[23] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating cross-platform development approaches for mobile applications," in *Proc. 8th WEBIST, Revised Selected Papers*, ser. Lecture Notes in Business Information Processing (LNBIP). Springer, 2013, vol. 140, pp. 120–138.

[24] H. Heitkötter and T. A. Majchrzak, "Cross-platform development of business apps with MD2," in *DESRIST*, ser. Lecture Notes in Computer Science, vol. 7939. Springer, 2013, pp. 405–411.

[25] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with MD^2," in *Proc. SAC '13*. ACM, 2013, pp. 526–533.

[26] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "MD2-DSL – eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen," in *ATPS '13*, ser. LNI, vol. 215. GI, 2013, pp. 91–106.

[27] H. Heitkötter, T. A. Majchrzak, U. Wolffgang, and H. Kuchen, *Business Apps: Grundlagen und Status quo*, ser. Working Papers. Förderkreis der Angewandten Informatik an der Westfälischen Wilhelms-Ulong-ersität Münster e.V., 2012, no. 4.

[28] H. Heitkötter, T. A. Majchrzak, B. Ruland, and T. Weber, "Evaluating frameworks for creating mobile Web apps," in *Proc. 9th WEBIST*. SciTePress, 2013, pp. 209–221.

[29] "HTML5," 2014, http://www.w3.org/TR/html5/.

[30] "Introducing Swift," 2015, https://developer.apple.com/swift/.

[31] H. Irawan, "Generation gap pattern vs protected regions in Xtext MDD," 2011,

http://eclipsedriven.blogspot.de/2011/10/generation-gap-pattern-vs-protected.html.

[32] W. Jeon, J. Kim, Y. Lee, and D. Won, "A practical analysis of smartphone security," in *Proc. Int. Conf. on Human Interf. and the Mgt of Inf.*, ser. HI'11. Berlin, Heidelberg: Springer, 2011, pp. 311–320.

[33] X. Jia and C. Jones, "AXIOM: A model-driven approach to cross-platform application development," in *Proc. 7th ICSOFT*, 2012.

[34] O. Le Goaer and S. Waltham, "Yet another dsl for cross-platforms mobile development," in *Proc. of the First Workshop on the Globalization of Domain Specific Languages*. ACM, 2013, pp. 28–33.

[35] Lee and B. Ware, *Open Source Development with LAMP: Using Linux, Apache, MySQL and PHP*. Boston, MA, USA: Addison-Wesley, 2002.

[36] T. A. Majchrzak, J. Ernsting, and H. Kuchen, "Model-Driven Cross-Platform Apps: Towards Business Practicability," in *Proc. of the 27th Conference on Advanced Information Systems Engineering (CAiSE) Forum*. CEUR, 2015.

[37] T. A. Majchrzak and H. Heitkötter, "Development of mobile applications in regional companies: Status quo and best practices," in *Proc. 9th WEBIST*. SciTePress, 2013, pp. 335–346.

[38] P. Miravet, I. Marín, F. Ortín, and A. Rionda, "DIMAG: A framework for automatic generation of mobile applications for multiple platforms," in *Proc. of the 6th International Conference on Mobile Technology, Application & Systems (Mobility)*. New York, NY, USA: ACM, 2009, pp. 23:1–23:8.

[39] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (Mobicom)*. New York, NY, USA: ACM, 2012, pp. 317–328.

[40] B. Myers, S. E. Hudson, and R. Pausch, "Past, present and future of user interface software tools," *ACM T COMPUT-HUM INT*, vol. 7, pp. 3–28, 2000.

[41] J. Ohrt and V. Turau, "Cross-platform development tools for smartphone applications," *IEEE Computer*, vol. 45, no. 9, pp. 72–79, 2012.

[42] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. New York, NY, USA: ACM, 2012, pp. 29–42.

[43] "PhoneGap," 2015, http://phonegap.com/.

[44] D. Prasanna, *Dependency Injection: Design Patterns Using Spring and Guice*. Manning Pub, 2009.

[45] A. Quilligan, "HTML5 Vs. Native Mobile Apps: Myths and Misconceptions," 2013, http://www.forbes.com/sites/ciocentral/2013/01/23/html5-vs-native-mobile-apps-myths-and-misconceptions/.

[46] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly, 2008.

[47] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley, 2005.

[48] M. Schulte and T. A. Majchrzak, "Context-dependent testing of apps," *Testing Experience*, pp. 66–70, September 2012.

[49] T. Stahl and M. Völter, *Model-driven software development*. Wiley, 2006.

[50] "Twitter acquires team behind visual app creation tool Cabana," http://thenextweb.com/twitter/2012/10/16/twitter-acquires-mobile-visual-app-creation-tool-cabana/.

[51] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco: Morgan Kaufmann, 2007.

[52] J. Vlissides, *Pattern Hatching: Design Patterns Applied*. Addison Wesley, 1999.

[53] S. Walderhaug, M. Mikalsen, G. Hartvigsen, E. Stav, and J. Aagedal, "Improving systems interoperability with model-driven software development for healthcare," *Stud Health Technol Inform*, vol. 129, pp. 122–126, 2007.

[54] J. Wargo, "SAP Mobile Platform: Cross-platform app development," 2014, http://scn.sap.com/community/mobile/blog/2014/09/26/sap-mobile-platform-cross-platform-app-development.

[55] W. West and S. M. Pulimood, "Analysis of privacy and security in HTML5 web storage," *J. Comput. Sci. Coll.*, vol. 27, no. 3, pp. 80–87, Jan. 2012.

[56] J. Whittle and J. Hutchinson, "Mismatches between industry practice and teaching of model-driven software development," in *Proceedings of the 2011th International Conference on Models in Software Engineering (MODELS)*. Berlin: Springer, 2012, pp. 40–47.

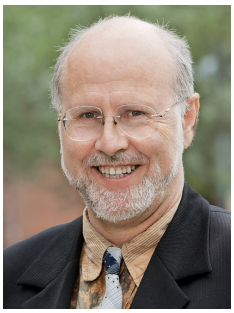
[57] "Xtext," 2015, http://www.eclipse.org/Xtext/.

[58] N. Zang, M. B. Rosson, and V. Nasser, "Mashups: Who? What? Why?" in *CHI '08 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2008, pp. 3171–3176.

AUTHOR BIOGRAPHIES

Dr. Tim A. Majchrzak is an associate professor for Information Systems at the University of Agder, Kristiansand, Norway. He received BSc and MSc degrees in Information Systems and a PhD in economics (Dr. rer. pol.) from the University of Münster. His research comprises both technical and organizational aspects of software engineering, often in the context of Web technologies and Mobile Computing. He has also published work on several interdisciplinary Information Systems topics.

Jan Ernsting is a researcher for Practical Computer Science at the Institute for Information Systems at the University of Münster, Germany. He received BSc and MSc degrees in Information Systems from the University of Münster. Jan is interested in Model-Driven Development of Mobile Apps as well as Development Processes enabling it.



Dr. Herbert Kuchen is professor for Practical Computer Science at the Institute for Information Systems at the University of Münster, Germany. He received his Diploma in Computer Science as well as his PhD and habilitation from RWTH Aachen University, Germany. He is interested in Programming Languages and Software Engineering in general as well as in Model-Driven Development and Development of Mobile Apps in particular.