Model-Driven Cross-Platform Apps: Towards Business Practicability

Tim A. Majchrzak[1], Jan Ernsting[2], and Herbert Kuchen[2]

[1] ERCIS, University of Agder, Kristiansand, Norway
[2] ERCIS, University of Münster, 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 provide 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 introduce MD^2 as one solution to fulfill typical requirements of business apps. Moreover, we highlight a business-oriented enhancement that further increases its business practicability.

Keywords: Cross-platform, MDSD, app, business app, mobile

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 with them (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) [20]. Besides some other topics such as security [9] and testing [24], cross-platform development is a major concern [14].

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 (cf. e.g. [12]) there are at least four major platforms 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 programming language, libraries, and usage of device-specific hardware – to name just a few factors. Developing separately for each platform currently is the choice; it is an error-prone and extremely inefficient procedure which becomes particularly frustrating when updating existing apps.

Based on the proliferation of adequate frameworks [17], mobile Webapps have become very popular. Cross-platform approaches based on Web technology such

as Apache Cordova [1] (a.k.a. PhoneGap [22]) 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 [6].

When working with Web apps, their origin in Web technology cannot be fully neglected. This has been called an *uncanny valley* [10]: a typical Webapps' look & feel is almost real but the app is slightly less responsive. Moreover, even with HTML5 [18] not all device-specific features are supported. Connecting to server-backends, as required for most *business apps* [16], is cumbersome and typically 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 [25] of apps).

This paper is structured as follows. Section 2 discusses different approaches that can be used for cross-platform development. MD^2, our approach to develop cross-platform apps, is introduced in Section 3. Section 4 characterizes the particularities of a business-oriented enhancement for MD^2. In Section 5, we draw a conclusion and sketch the path for MD^2.

2 Existing Approaches

Cross-platform app development approaches have been discussed as early as 2009. Miravet et al. present their framework DIMAG, which is based on State Chart eXtensible Markup Language (SCXML) [21]. Even what is considered a cross-platform approach might vary – practitioners sometimes employ loser definitions (cf. the comparison by [7]). Apache Cordova [1] utilizes Web technology but also supports accessing native device features. Yet, its Web foundations negatively impact aspects such as app responsiveness. Other approaches build upon a *self-contained runtime* operating on custom scripting languages such as Titanium [2].

Cabana [8], AXIOM [19], and applause [3] are *generative approaches*. Cabana focuses on app development in the context of higher education. It utilizes a GUI to manipulate graphic models of app representations and allows to implement customized code. However, interactions with backends are neglected and using other platforms for achieving this is advised (cf. [8, p. 533f.]). Cabana apparently has been discontinued [26]. AXIOM takes a technical stance as it features aspects of UML and uses the programming language Groovy [13]. Moreover, it does not fully automate intermediate steps of code generation [15]. applause is most similar to MD^2: it provides a DSL, too. Yet, it is mostly restricted to displaying information and does not provide a DSL tailored to describing business apps.

3 Cross-Platform Development with MD²

3.1 Introductory Example

A corporation has a customer relationship management (CRM) system that it wants to provide access to its sales representatives so that they can 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 MD² to integrate its CRM and the mobile apps that are to be created. The scope of the example is limited to keep it brief: it will exclusively focus on recording prospective customers through mobile apps.

We now successively introduce MD²'s architecture, features, and domain specific language.

3.2 Architecture and Features of MD²

Using the textual MD²-DSL the corporation defines a model. From that MD² model the artifacts in the shaded area of Figure 1 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.
- Generated MD² 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 MD² backend to link it to the corporation's CRM API (corresponds to the link from MD² to the CRM in Figure 1). This typically is a straightforward task.

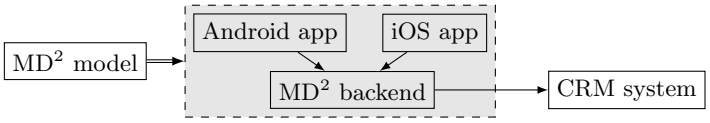

Fig. 1. Basic Architecture

3.3 MD²-DSL

A MD²-DSL model is structured according to the well-known *model-view-controller* pattern [5]. Thus, it consists of three parts specifying the model, view, and controller component of an app. In our example, the model (see Listing 1.1) defines just a single entity type `Contact` (with first name, surname, etc.) and an enumeration type `AcquisitionState`.

```
entity Contact {
  firstname : string
  surname : string
  phone : integer (optional)
  email : string (optional)
  state : AcquisitionState
}

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

Listing 1.1. MD^2 model

As depicted in Listing 1.2, 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 1.3 – 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. In Figure 2, the view that is generated from the view definition is shown.

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

Listing 1.2. MD^2 view definition

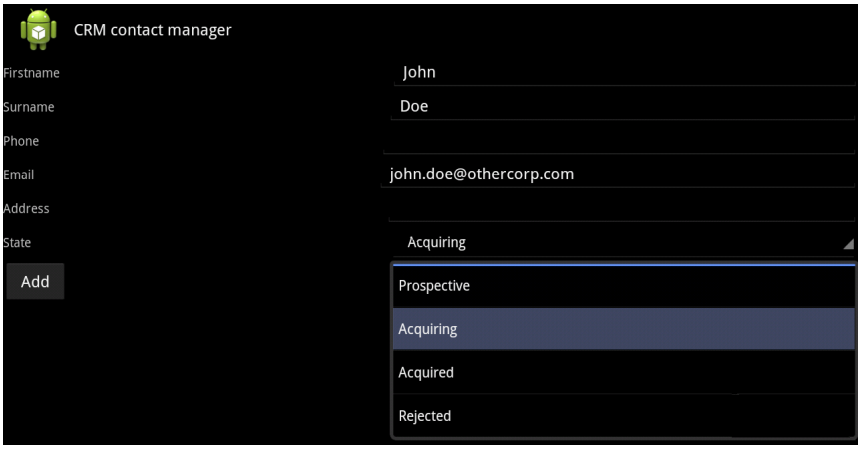

Fig. 2. Generated Android view

The corresponding controller component (see Listing 1.3) 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 1.3, 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.

```
1  main {
2    appName "CRM_contact_manager"
3    appVersion "1.0"
4    modelVersion "1.0"
5    startView addContactView
6    onInitialized startUpAction
7  }
8
9  contentProvider Contact contactContentProvider {
10   providerType crmSystem
11 }
12
13 remoteConnection crmSystem {
14   uri "http://md2.crm.corp.com/"
15 }
16
17 action CustomAction startUpAction {
18   bind action addAction on addContactView.addButton.onTouch
19 }
20
21 action CustomAction addAction {
22   call DataAction (save contactContentProvider)
23 }
```

Listing 1.3. MD^2 controller definition

3.4 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.

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 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). Nevertheless, you could consider MD^2 as "one-DSL-fits-most-cases".

While it might be undesirable to add custom code on the frontend (i.e. the app – the MDSD approach would become blurred), it would be helpful to have improved

support for backend customizations. Possibly suitable approaches discussed in the literature are the generation gap pattern [11, pp. 571ff.], protected regions [25, p. 29], and dependency injection [23].

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. [4]). Nevertheless, we have not yet addressed testing explicitly.

As a final remark, there are no specific security features built into MD^2. Due to the DSL, it is hardly possible to use MD^2 maliciously anyway and business logic typically resides on the backend. 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.

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.

Regarding our previous example, sales representatives need to get access to customer records as well as previous interactions. For that, the entity type `Contact` in Listing 1.4 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 1.5). 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.

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

Listing 1.4. Augmented MD^2 model

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

Listing 1.5. Augmented MD^2 view definition

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 paper, 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 Conclusion

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, testing of MD^2's components (preprocessors, generators, etc.) was neglected so far. On a more user-centric level, testing of MD^2 apps could also be relevant for companies but also for a broader non-MD^2 related audience as well. Improving testability for MD^2 also allows providing stable artifacts (i.e. development tools, plugins, 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 we will address.

Even though native code for the two most common platforms is generated, 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.

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.

References

1. Apache Cordova (2014), http://cordova.apache.org/
2. Appcelerator (2014), http://www.appcelerator.com/
3. applause (2014), https://github.com/applause/
4. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented Software Architecture: A System of Patterns. Wiley, New York, NY, USA (1996)
6. Charland, A., Leroux, B.: Mobile application development: web vs. native. Commun. ACM 54, 49–53 (2011)
7. Cowart, J.: "Pros and cons of the top 5 cross-platform tools", http://www.developereconomics.com/pros-cons-top-5-cross-platform-tools/
8. Dickson, P.E.: Cabana: a cross-platform mobile development system. In: Proc. 43rd SIGCSE. pp. 529–534. ACM (2012)
9. Dye, S.M., Scarfone, K.: A standard for developing secure mobile applications. Comput. Stand. Interfaces 36(3), 524–530 (Mar 2014)
10. Fowler, M.: CrossPlatformMobile (2011), http://martinfowler.com/bliki/CrossPlatformMobile.html
11. Fowler, M.: Domain-Specific Languages. Addison-Wesley Pearson Education (2011)
12. Gartner Press Release (2012), http://www.gartner.com/it/page.jsp?id=1924314
13. Groovy (2014), http://groovy.codehaus.org/
14. Heitkötter, H., Hanschke, S., Majchrzak, T.A.: Evaluating cross-platform development approaches for mobile applications. In: LNBIP, vol. 140, pp. 120–138. Springer (2013)
15. Heitkötter, H., Majchrzak, T.A., Kuchen, H.: Cross-platform model-driven development of mobile applications with MD^2. In: Proc. SAC '13. pp. 526–533. ACM (2013)
16. Heitkötter, H., Majchrzak, T.A., Wolffgang, U., Kuchen, H.: Business Apps: Grundlagen und Status quo. No. 4 in Working Papers, Förderkreis der Angewandten Informatik an der WWU Münster e.V. (2012)
17. Heitkötter, H., Majchrzak, T.A., Ruland, B., Weber, T.: Evaluating frameworks for creating mobile Web apps. In: Proc. 9th WEBIST 2013. pp. 209–221. SciTePress (2013)
18. HTML5 (2014), http://www.w3.org/TR/html5/
19. Jia, X., Jones, C.: AXIOM: A model-driven approach to cross-platform application development. In: Proc. 7th ICSOFT (2012)
20. Majchrzak, T.A., Heitkötter, H.: Development of mobile applications in regional companies: Status quo and best practices. In: Proc. 9th WEBIST. pp. 335–346. SciTePress (2013)
21. Miravet, P., Marín, I., Ortín, F., Rionda, A.: DIMAG: A framework for automatic generation of mobile applications for multiple platforms. In: Proc. Mobility '09. pp. 23:1–23:8. ACM, New York, NY, USA (2009)
22. PhoneGap (2014), http://phonegap.com/
23. Prasanna, D.: Dependency Injection. Manning Pub (2009)
24. Schulte, M., Majchrzak, T.A.: Context-dependent testing of apps. Testing Experience pp. 66–70 (September 2012)
25. Stahl, T., Völter, M.: Model-driven software development. Wiley (2006)
26. "Twitter acquires team behind visual app creation tool cabana", http://tnw.to/e67X