Refining a Reference Architecture for Model-Driven Business Apps

Jan Ernsting[1], Christoph Rieger[1], Fabian Wrede[1] and Tim A. Majchrzak[2]
[1]*University of Münster, Münster, Germany*
[2]*University of Agder, Kristiansand, Norway*

Keywords: Reference Architecture, MDSD, App, Mobile, Mobile App, Business App, Architecture.

Abstract: Despite much progress, cross-platform app development frameworks remain a topic of active research. While frameworks that yield native apps are particularly attractive, their spread is very limited. It is apparent that (theoretical) technological superiority needs to be accompanied with profound support for developers and adequate capabilities for maintaining the framework itself. We deem so called reference architectures to be a major step for building better cross-platform app development frameworks, particularly if they are based on techniques of model-driven software development (MDSD). In this paper, we describe a refinement of a reference architecture for business apps. We employ the model-driven cross-platform development framework MD^2 for this purpose. Its general design has been described extensively in the literature. The framework has a sound foundation in MDSD, yet lacks a generator support that fulfils the above sketched goals. After describing the required background, we argue in detail for a suitable reference architecture. While it will be a valuable addition to the MD^2 framework, the discussion of our findings also makes a contribution for generative app development in general.

1 MOTIVATION

Cross-platform development frameworks for apps have gained much popularity (Ohrt and Turau, 2012; Holzinger et al., 2012). However, most of them employ Web technology and yield inferior results when striving for a *native look and feel* (Joorabchi et al., 2013; Heitkötter et al., 2013a). Creating cross-platform frameworks is a profound technological challenge (cf. (Heitkötter et al., 2013b)). At the same time, a framework needs to be comprehensible for developers. Moreover, it should provide adequate development support. The latter is particularly true when a framework seeks to improve the app development process in general (cf. (Heitkötter et al., 2015)).

In this paper, we build on previous work on MD^2, a cross-platform development framework that provides fully native apps for the supported target platforms. It employs techniques from model-driven software development (MDSD). Apps are rather modelled than programmed; for this purpose, a domain-specific language (DSL) is used, which has been tailored to business app development (Heitkötter et al., 2013c).

Despite its undoubted technological soundness, MD^2 has not yet found widespread adoption. This without question can be attributed to a missing community and user base (i.e. economies of scale). We believe that part of the reason lies in the complexity of generation (Evers et al., 2016), though. Code generation might seem negligible from an app developer's point of view since capabilities are taken for granted when developing a specific app. However, improvements in the generation step are also reflected in development in form of extended capabilities and the possibility to benefit from improvements to the DSL. We have, therefore, argued to overcome the existing challenges with profound work on the app generation step (Evers et al., 2016).

With a more detailed look at the problem, additional questions need to be asked:

- Can a design pattern such as Model-View-Controller (MVC) or Model-View-ViewModel (MVVM) (Smith, 2009) be implemented throughout the development process?
- How can the fragmentation of devices and the heterogeneity of software be overcome?
- How can frequent releases of platforms (such as Android and iOS), changes to the underlying programming languages (such as Apple's transitioning from Objective-C to Swift (Swift Blog, 2015)), and modifications to the ecosystems be effectively reflected in the generators?

ISBN: 978-989-758-186-1

- How can redundancy in a typical MVC description of apps be avoided?

Even though generators are seldom implemented from scratch and only updated occasionally, their development demands great skill. Code generators facilitate the embedding and have to account for two driving forces: a model instance on the one side and the target platform on the other. With this paper, we refine our previous work (Evers et al., 2016). This refinement builds on two shortcomings that we perceived:

- Currently, MD^2 focuses on object structure and behaviour, and not on interaction.
- The employed top-down approach (from model to reference architecture to platform-specificity) does not take into consideration platform-specific features in an effective fashion.

Therefore, this paper presents a more effective reference architecture that advanced from the feedback to realising a previously proposed reference architecture with two distinct platforms. Thereby, it greatly helped to fix the ecosystem under test except for the code generation stage. We were thus enabled in assessing and revising the reference architecture accordingly.

The main contributions of this paper are twofold. Firstly, we propose a detailed, sophisticated reference architecture for the model-driven creation of business apps. While it has been tailored to use in MD^2, it is applicable to MDSD for apps (as well as in Web technology-based frameworks) in general. Secondly, we discuss our findings. This further increases the generalizability of our work.

This paper is structured as follows. Section 2 draws the background of related work on MDSD for business app development and the corresponding use of reference architectures. Section 3 then presents an evaluation of the existing reference architecture to provide the foundation for the new proposal. This is presented as a detailed refinement in Section 4. Our findings are then discussion in Section 5. Finally, we draw a conclusion in Section 6.

2 RELATED WORK

Apart from business apps and related work in the area of cross-platform development approaches, this section specifically highlights a concrete approach for business apps, namely MD^2. In addition, the general area of reference architectures is considered to base the refinements on.

2.1 Business Apps and App Development

Besides common apps for purposes such as social networking or entertainment, business apps encompass those that are directed at interacting with businesses' backend information systems. In addition, they can be categorised by their form-based and data-driven nature (Majchrzak et al., 2015).

For apps, in general, to overcome the heterogeneity of mobile platforms, various approaches exist. Their spectrum ranges from Web apps to native apps (Majchrzak et al., 2015). While the former rely on Web technologies such as HTML5, Cascading Style Sheets (CSS), and JavaScript, the latter requires sufficient proficiency of the respective platforms, the applicable programming languages, and the respective software development kits (SDKs). Generative approaches aim at providing a native look and feel as well as native performance while striving to be as convenient to use as Web-based frameworks (Heitkötter et al., 2013a). There are two kinds of generative cross-platform frameworks: *transpilers* and *MDSD*-based approaches.

Transpilers allow compiling source, intermediate, or binary code for one platform to another. However, current transpilers such as J2ObjC (J2ObjC, 2015) deliberately focus on non-UI code. Therefore, they do not support a complete transformation and require at least some finishing touches to let apps run on targeted platforms. Consequently, they are neither widely used nor can be considered to be particularly mature.

Model-driven generative approaches employ the techniques of MDSD (Stahl and Völter, 2006). Typical examples for approaches encompass commercial products such as WebRatio (WebRatio, 2015). It utilises graphical models that are expressed in the Interaction Flow Modeling Language (IFML), which the backing company pushed to standardisation by the Object Management Group (OMG). Though restricted to displaying data, applause (applause, 2015) features a textual domain-specific language (DSL) to express models in. While applause's development was discontinued for more than a year and Xmob (Le Goaer and Waltham, 2013) has yet to present concrete modelling facilities, development on MD^2 progressed.

2.2 Creating Business Apps with MD^2

MD^2 is based on a textual DSL that is structured along the model-view-controller (MVC) design pattern (Gamma et al., 1995; Buschmann et al., 1996).

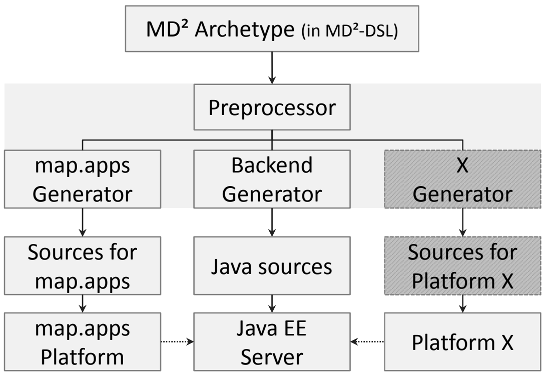

Figure 1: MD^2 Modelling Process.

To overcome platform heterogeneity, its syntax focuses on describing business apps. Thus, consolidated abstractions are used to express archetypes in MD^2-DSL (Heitkötter et al., 2013b). This has obvious limitations, as featured elements correspond to the lowest common denominator of platform features. This is particularly true for view related aspects (cf. Section 4). On the other hand, users benefit from business apps automatically derived from archetypes without needing in-depth knowledge of target platforms. In consequence, users have to weigh the pros and cons of using a model-driven approach.

To mince matters, approaches can boost their usefulness by increasing the number of supported platforms. This is achieved by adding code generation facilities for desired platforms. An archetype passes through a preprocessor as shown in Figure 1. That stage augments the input archetype with concrete elements to reduce generation effort (Majchrzak and Ernsting, 2015). Next, the augmented archetype is handed over to code generators. Currently, these output sources for the commercial map.apps platform (map.apps product description, 2015) and a Java EE based backend. Nevertheless, the modelling process incorporates provisions for adding generators such that these dispense applications for arbitrary platforms such as Windows Phone or Symbian.

Developing a code generator requires its engineers to comprehend the archetype's structure (i.e. the metamodel and its instances) as well as the targeted platform ecosystem (i.e. the programming environment, common libraries, etc.). Thus, engineers have to account for these two forces that drive generator development. In the following, assistive means that reduce the overall development effort are illustrated.

2.3 Reference Architectures

To understand the relevance of developing and further improving a reference architecture for MD^2 apps, knowledge of the potential benefits of reference architectures is advisable. This question was already subject of academic research; however, it was discussed in a broader context and not specifically tailored to model-driven software development and mobile apps in particular (cf. (Cloutier et al., 2010; Angelov et al., 2009)). Nonetheless, the findings can be transferred to this area.

First of all, as any other form of architectures, a reference architecture helps to control complexity (Cloutier et al., 2010). The design of software is expressed in a standardized form such that it is easily understandable for developers. However, in the context of MD^2, a reference architecture main benefit consists in the preserved knowledge and guarantees a common understanding. The reference architecture is supposed to build a common ground on which the development of generators for new platforms takes place. Consequently, it has to embrace the knowledge and insights which were gained during the implementation of other generators and applications in order to provide these to its consuming developers.

A common understanding of MD^2 application components is necessary to keep the effort for development and maintenance of existing generators manageable. The architecture of the generators and applications should be similar to a certain degree. This facilitates a quick understanding of the concrete architecture on different platforms. Thus, developers who implemented a generator for a certain platform could perform maintenance tasks on other platform generators, as the underlying concepts are the same.

3 EVALUATION OF A REFERENCE ARCHITECTURE FOR MODEL-DRIVEN BUSINESS APPS

Various best practices and established patterns for mobile platforms exist. Unsurprisingly, these typically neglect code generation as they do not distinguish between model-specific elements and components concerned with the general application execution.

However, in the context of model-driven approaches, the aforementioned code generation stage creates a tie between the model and its targeted platforms. Thus, the generation stage forms a crucial component of these approaches.

When considering MD^2's use of the MVC pattern in its DSL model, sufficient guidance with regard to engineering apt code generators could be expected. Yet, previous efforts showed that redundant or recurring ideas could not be removed by the MVC pattern itself nor did the tool's underlying metamodel offer substantial guidance in developing code generators. MD^2's constitutional development had focused on Android and iOS as target platforms. At that stage, its engineers noticed to some degree that numerous architectural choices arose for both platforms (Evers et al., 2016).

Prior to targeting a third, previously unconsidered commercial platform, (Evers et al., 2016) compiled a reference architecture for model-driven business apps. Establishing support for map.apps (map.apps product description, 2015) not only illustrated the reference architecture's applicability, but also showed that MD^2's scope could surpass the limits of mobile platforms.

The reference architecture explicates dependencies between and usages of MD^2-DSL's metamodel elements. On a coarse level, the MVC pattern alone did not facilitate development of code generators in an intuitive fashion. Nevertheless, architectural elements can easily be partitioned to accentuate MD^2's MVC provenance as shown in Figure 2. To provide guidance for generator development, (Evers et al., 2016) described architectural key elements with regard to their instantiation and runtime behaviour.

For example, concrete `EventHandler` account for different event types that exist in MD^2. These architectural elements assist in orchestrating an app's runtime behaviour. They determine which associated `Action` has to be executed in response to triggered events. Here, the architecture's explication guides code generators to provide facilities to handle global (e.g. context related) events such as connection lost, view related (i.e. widget) events, and data persistence (i.e. content provider) events in their respective generates (i.e. apps).

Of course, (Evers et al., 2016) elaborated on the other elements in Figure 2. For now, we highlight their discussion in which they point out that little backing empiricism exists and that their architecture may require minor changes to be universally applicable to other platforms. Put straight, we strove to evaluate the architecture's suitability by targeting the two platforms currently dominating the mobile market: iOS and Android (Gartner, 2015).

The implementation was conducted as a greenfield approach, respecting the development process proposed by (Stahl and Völter, 2006, p. 27). Two independent developers, who were not involved in the development of the original reference architecture, implemented the reference apps and generators for the corresponding platforms, leveraging the original reference architecture as a guideline for the respective concrete architectures.

The evaluation of the reference architecture revealed that it was already quite well suited to provide guidance regarding the structural composition of required components. However, overall it was difficult to instantiate the reference architecture because of two main shortcomings: first, the reference architecture lacks to provide an overview of the runtime behaviour and the interaction between components and second, it chooses a misleading level of abstraction at some points, thus slackening the platform-specific development progression. Therefore, the reference architecture was revised and applied as presented in the following.

4 REVISING THE REFERENCE ARCHITECTURE

Because of the aforementioned shortcomings, structural aspects of the original reference architecture were improved and developers relieved with increased flexibility regarding the implementation in programming code as well as increased clarity towards interaction patterns.

4.1 Reference Architecture Structure

The revised reference architecture improves the structure three-fold: first, the progression of the MD^2 language introduced a process layer extension that is reflected in the current architecture version. Furthermore, the previously unspecified view layer of the application is substantiated and a mobile-centric task queue component is added.

4.1.1 Workflow Layer

Since the initial development of the reference architecture, the MD^2 language further evolved. Most prominently, an additional layer of so-called workflow elements was added to its specification, enabling development of cross-app workflows and app product lines (Dageförde et al., 2016). This process-oriented layer is now also considered in the revised reference architecture. For a seamless integration, the current building blocks of events and actions were reused. To trigger a workflow event, an action called `SetWorkflowElementAction` extends the existing architecture. Newly added `WorkflowElement` objects represent self-contained process steps. Every workflow element is supplied with a workflow *event*

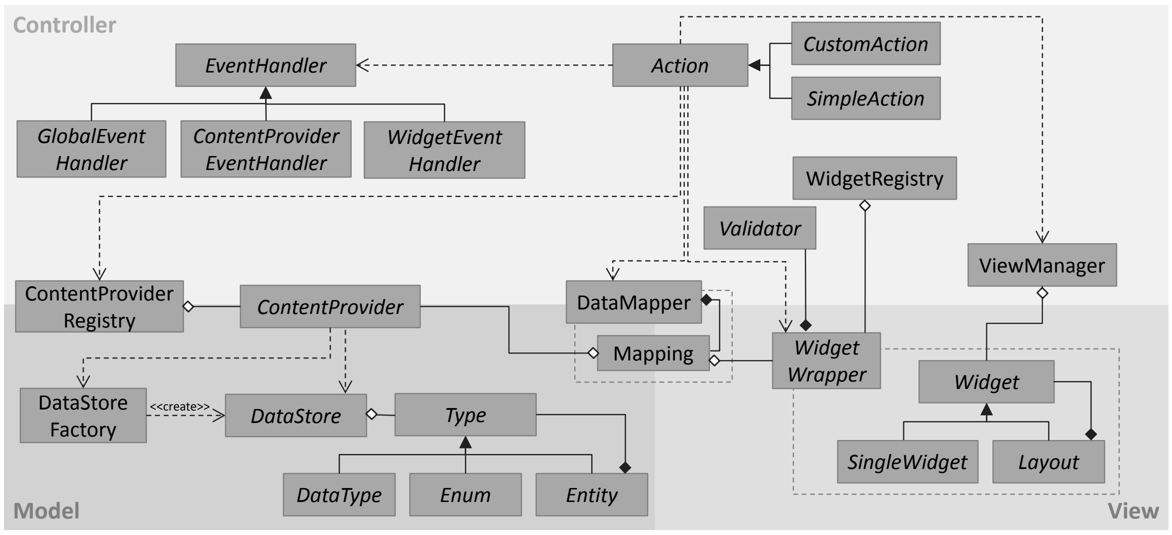

Figure 2: Original Reference Architecture (Evers et al., 2016).

to identify transition points within the overall process. Also, a workflow action indicates whether to start or end the specified workflow. Together, conditional workflow *paths* can be modelled.

The actual processing of the workflow elements is performed by the `WorkflowManager` component. For local workflow elements, this includes finalizing the currently active workflow element, deciding upon the further workflow path, and starting the next element. In case the workflow element has to be continued in another app, the intermediate state of the workflow and its associated entities is additionally sent to the backend server. The `WorkflowManager` component of the subsequent app is then notified about a newly available workflow instance and app users can eventually continue its execution.

4.1.2 View Specification

The original version of the reference architecture did not consider the implementation of the view as a part of the architecture itself. It was claimed that the implementation of the view was too platform-specific and, therefore, could not be included. However, our evaluation of the reference architecture on iOS and Android showed that there are similarities regarding the structure of views. That is on both platforms, the view is defined as a hierarchy of layouts and widgets that can be arbitrarily nested. This pattern is also common on other platforms. Therefore, objects representing the structure of the view were included in the reference architecture. In case that a platform handles the view differently, this part could easily be adapted when transforming the reference architecture into a platform-specific one.

The `WidgetWrapper` object was removed in the revised reference architecture. At this point, the original reference architecture stated that a widget should provide a certain set of methods. However, this was already done by applying a specific design pattern, which is not necessarily the best choice for the concrete implementation. On Android, for example, it turned out to be more convenient to use Custom Views instead of wrapper objects. Therefore, we claim that the application of concrete design patterns should be avoided in order not to push developers in a direction that might not be the best choice. Rather, the reference architecture should be more general and show options on how to develop the concrete implementation.

4.1.3 Tasks

In MD^2, custom actions consist of atomic tasks. These tasks perform operations such as binding the value of a widget to an entity. Tasks were not a part of the original reference architecture as they were supposed to be directly transformed into plain code by app generators. However, the evaluation showed that tasks can still be further generalized. Consequently, it is beneficial to add classes for the different tasks in the platform-specific libraries and to instantiate them with required parameters in the generated code. Tasks differ from actions in a way such that actions perform high-level, control flow-oriented operations, such as switching to another view, whereas tasks perform low-level, view-oriented operations, such as enabling data binding to a widget.

4.1.4 TaskQueue

In general, mobile applications should be designed in a resource efficient way as mobile devices typically provide limited resources compared to personal computers. The Android platform, for example, provides a built-in memory management that is enabled to free up memory allocated to paused activities so that it can be used by activities with a higher priority, e.g. active ones. Consequently, it is not guaranteed that widget objects exists all the time an app is running. However, for certain tasks, such as data binding, it is necessary to have access to a widget, e.g. to register event listeners. The first version of the reference architecture suggested overcoming that issue by creating `WidgetWrappers` on start-up of the application. As these `WidgetWrappers` keep a reference to the actual widget, the Android memory management does not destroy the objects so that they can be accessed if required during the execution of the app. However, this approach leads to memory leaks, i.e. objects that blocked memory and could not be deleted, because they were referenced by a running app. To avoid these leaks and to facilitate a more dynamic handling of application resources, the concept of the `TaskQueue` is introduced in the revised version of the reference architecture.

The `TaskQueue` provides functionality to store tasks that cannot be executed because of required objects missing. As soon as the required objects are created, the execution is triggered again. An example from the Android platform is the transition between activities. When a new activity is started, all widget objects that belong to the activity's view are created. Therefore, it might be possible to execute tasks that could not be executed before. Thus, the transition between activities is one point in the flow of an Android app where the execution of formerly stored tasks should be triggered again.

4.2 Platform-specific Implementation Variability

Possible alternatives for the implementation of the architecture in object-oriented programming languages include the approaches that are discussed in the following.

4.2.1 Status Quo

The original reference architecture was developed with a top-down approach starting from the MD^2 language definition. As a consequence, platform-specific characteristics were mainly treated as limitations that needed to be bypassed by additional generalised components. For example, the event handling mechanism is designed as explicit component because platforms not necessarily provide the possibility to extend their native event system.

As further contribution, the revised architecture incorporates the implementation learning in a bottom-up manner. Particularly, the main pain point of over-generalisation, resulting in tedious re-implementations of existing platform features, should be avoided. Mapping the reference architecture to appropriate platform implementations is a problem to be solved by generator developers with knowledge of the target platform's characteristics and advanced language constructs. This choice of implementation allows for the necessary variability to avoid implementation overhead and benefit from available language features. (Fazal-e-Amin et al., 2011) discuss several techniques for object-oriented programming languages that are applicable for the MD^2 reference architecture.

4.2.2 Delegation

Glue interfaces bridge potentially incompatible code parts by specifying an interface towards the rest of the application (France and Rumpe, 2007). As a basis, all components of the revised reference architecture can be regarded as such interfaces instead of explicit class requirements. Components that cannot easily be specified as one object on the target platform can use available aggregation or delegation mechanisms to perform the desired action as long as the interface towards the remaining objects is fulfilled. Developers can therefore benefit from unique platform features to reduce development time, increase runtime performance, or improve maintainability and readability of the resulting code.

For example, the implementation of data mappings depends on available platform features. Where possible, an efficient bidirectional map of model entity and view element is a good solution. Other implementations are likely suitable as long as the data mapper can be queried to look up the respective widget to an entity attribute, and vice versa.

4.2.3 Inheritance

Extending classes of the platform is a viable solution to reuse existing functionality contained in the platform libraries. Using class inheritance mechanisms, provided classes only need to be enriched with missing operations in order to comply with the reference architecture specification, again reducing implementation efforts.

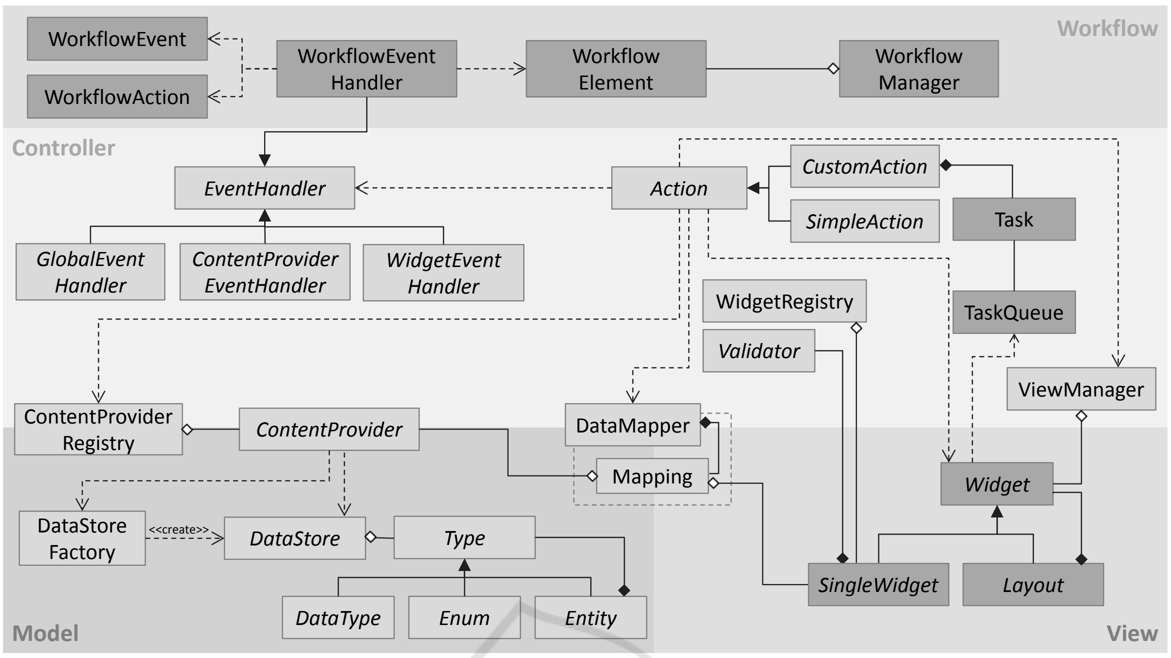

Figure 3: Revised Reference Architecture.

Native widget extensions are a possible application of this approach: Platforms such as Android support the creation of custom widgets by sub-classing a generic widget class. Consequently, only additional functionalities for validation and value access need to be implemented manually. Alternatively, wrapper objects can be used that refer to native widget elements as fallbacks. As a second option, the native platform event system can be extended with custom events on some platforms; thus limiting implementation overhead. Where this is not possible, for instance on iOS, all event-related components of the reference architecture need to be implemented.

4.2.4 Overloading

Overloading methods or using generic classes is a further technique to flexibly implement the required methods of the reference architecture's component specifications. By choosing appropriate parameters, the desired functionality can be provided while leveraging the potential of the respective programming language concepts.

For instance, dynamically typed languages such as JavaScript might look up view element references using String objects. However, statically typed languages may benefit from enumeration types by providing additional compile-time security with regard to the existence of such references.

4.2.5 Decentralized Processing

Several components of the reference architecture manage other components. However, it is deliberately unspecified whether a single object should perform all management activities or whether responsibilities can be shared by a distributed set of instances. For example, event handlers can be implemented as singleton objects that handle specific events on a global application level. On the other hand, the Android implementation initializes individual event handlers that use the observer pattern to directly capture changes of the widgets and content providers (Gamma et al., 1995, p. 293).

As a result, the revised reference architecture combines the previous model-driven top-down approach with a platform-driven bottom-up perspective while at the same time encouraging implementation choices by generator developers.

4.3 Reference Architecture Interactions

To assist developers with respect to their implementation choices and expatiate on desired component interactions in the respective implementations, further behavioural aspects require clarification complementing the revised structure of the reference architecture. Particularly, three main interactions are essential for the application execution: basic widget control flows, process-oriented workflow control flows, and data flows.

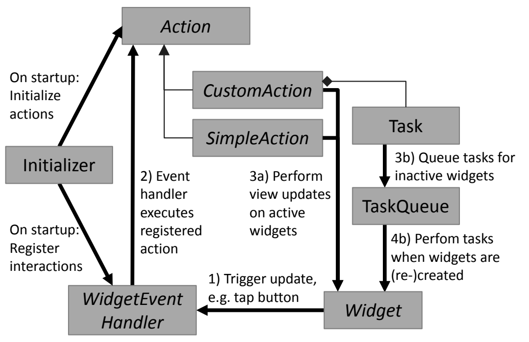
Action
CustomAction
SimpleAction
Task
On startup: Initialize actions
Initializer
2) Event handler executes registered action
3a) Perform view updates on active widgets
3b) Queue tasks for inactive widgets
TaskQueue
4b) Perfom tasks when widgets are (re-)created
On startup: Register interactions
1) Trigger update, e.g. tap button
WidgetEvent Handler
Widget

Figure 4: Interaction diagram for widget control flows.

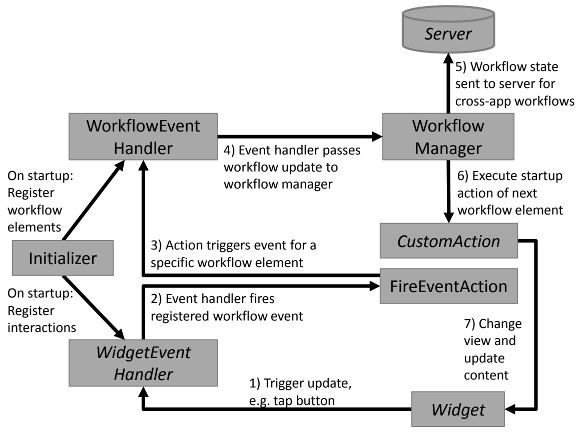
Server
5) Workflow state sent to server for cross-app workflows
WorkflowEvent Handler
4) Event handler passes workflow update to workflow manager
Workflow Manager
On startup: Register workflow elements
6) Execute startup action of next workflow element
Initializer
3) Action triggers event for a specific workflow element
CustomAction
FireEventAction
On startup: Register interactions
2) Event handler fires registered workflow event
7) Change view and update content
WidgetEvent Handler
1) Trigger update, e.g. tap button
Widget

Figure 5: Interaction diagram for workflow control flows.

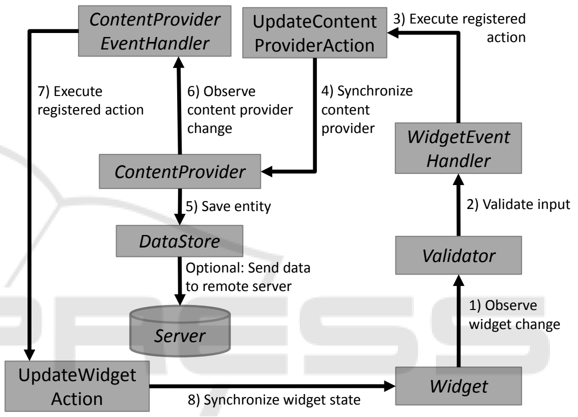
ContentProvider EventHandler
UpdateContent ProviderAction
3) Execute registered action
7) Execute registered action
6) Observe content provider change
4) Synchronize content provider
WidgetEvent Handler
ContentProvider
5) Save entity
2) Validate input
DataStore
Validator
Optional: Send data to remote server
1) Observe widget change
Server
UpdateWidget Action
8) Synchronize widget state
Widget

Figure 6: Interaction diagram for data flows.

4.3.1 Widget Control Flow

In MD^2, apps' business logic is modelled using MD^2 actions which, amongst others, can trigger view updates. Events and actions are created and mutually registered by the `Initializer` component which sets up all interacting objects on application launch and triggers the first event. From this point onwards, the event-action loop is the backbone of the application runtime. As depicted in Figure 4, widget changes are observed through respective gesture or value change events. Event handlers then execute the respective actions registered on start-up of the application. Depending on the state of the targeted view element, updates such as widget state changes or view transitions can directly be applied on visible elements. Update tasks on currently inactive elements, for instance data binding changes, are temporarily queued in the `TaskQueue` as described in Subsubsection 4.1.4.

4.3.2 Workflow Control Flow

In addition to this app-internal view update mechanism, the overall business process modelled as multiple workflow elements (Dageförde et al., 2016) also leverages the event-action loop concept. Instead of updating a view element, the widget event handler executes a `FireEventAction` that further triggers a workflow event (cf. Figure 5). The respective event is handled by the `WorkflowEventHandler` that notifies the `WorkflowManager` component to process the event as described in Subsubsection 4.1.1. When starting the next workflow element, its start-up action is executed which may contain model-specific initialization tasks. It also switches to the respective view and updates widget contents according to the regular widget control flow description.

4.3.3 Data Flow

As MD^2 apps are data-driven, data flows within the application are a major element of interest. Starting from observed widget changes again, first input validation is applied to avoid unnecessary data flows. Next, event handlers execute the registered action which notifies the content provider of an update. This content provider manages the underlying entity and saves the field to either local storage or a remote location accessed by a data store. Finally, the content provider also emits an update event such that the respective event handler executes update actions for (potentially multiple) associated widgets. This loop, depicted in Figure 6, can also be started when an external data source changes and the data store notifies the content provider of updated content.

These interaction loops additionally guide the behavioural implementation of the reference architecture on new platforms without enforcing any technical implementation approach with regard to the event and synchronization techniques.

5 DISCUSSION AND OUTLOOK

The revised reference architecture incorporates insights gained from the application of the original architecture in three distinct generator implementations. However, the focus on the MD^2 language still limits the empirical validation of the approach. On the one hand, the balance of component generalisation and flexibility of implementation derived from the evaluated generator implementations needs to be validated. On the other hand, the described interaction illustrations are also based on the currently available generator implementations. In retrospect, this information would have provided important guidance for the understanding of the reference architecture. Still, for both aspects empirical validation can only be achieved by applying the refined reference architecture to another platform *without* prior knowledge of the subject matter.

While the interaction patterns do not impose any restrictions on the actual implementation, there might also be platforms for which the basic event-action loop is not well applicable. With additional communication emerging from cross-app workflow coordination, it should be considered to further specify the behaviour of the existing external interfaces as guidance for generator developers. For instance, data exchange formats on mobile platforms should be assessed to provide further standardisation. Also, different communication mechanisms such as asynchronous backend requests or servers pushing messages to apps may be considered with regard to user experience improvements. Incorporating such changes into the reference architecture may result in changes to the interaction mechanisms that were presented in this paper.

Finally, the derived best practices need to be reapplied to the generator implementations fostering a maintainable code base following common architectural design decisions. The revised reference architecture already incorporates changes resulting from the MD^2 language evolution as for instance the workflow layer extension integrates nicely into the existing architecture. Yet, it has to be shown whether the current reference architecture is flexible enough to adapt to future DSL language changes.

Despite some minor drawbacks, the evaluation showed that reference architectures can serve as supportive means for extending model-driven approaches such as MD^2. In addition, these limitations were leveraged to assist in revisiting and applying these newly gained insights as presented in Section 4.

6 CONCLUSION

In this paper, we have presented work on the refinement of a reference architecture for MD^2. It extends the model-driven cross-platform framework with means to provide unified, maintainable, and scalable code generation. Thereby, it ultimately also contributes to the framework's ease-of-development.

Based on the study of related work and the first suggestion by (Evers et al., 2016), we proposed steps for the refinement. Despite some minor drawbacks, the evaluation showed that reference architectures can serve as supportive means for extending model-driven approaches such as MD^2. These limitations were leveraged to assist in revisiting and applying the newly gained insights. The actual work consists of detailed suggestions for the structure of the reference architecture, ways to address platform-specificity while keeping an abstract interface, and suggested guidelines for component interactions.

There is a fine line between being too specific and too general (or, rather, abstract) with regard to reference architectures. We are confident that we have found a balanced approach with the proposals made in this paper. However, the feasibility of our ideas will need to be proven empirically – first qualitatively and ultimately quantitatively. In the meantime, we will keep up our work and also seek to contribute to the core of the MD^2 framework including its domain-specific language. Currently, an alternative, graphical modelling front-end that utilises the revised reference architecture and generators is being designed to assess its accessibility to modellers; thus, making a case in favour of reusable and maintainable generation facilities. We hope that MD^2 will get more attention by industrial users in the future, probably even stimulating work on complementary or competing approaches.

REFERENCES

Angelov, S., Grefen, P., and Greefhorst, D. (2009). A classification of software reference architectures: Analyzing their success and effectiveness. In *European Conf. on Software Architecture (ECSA)*, pages 141–150. doi: 10.1109/WICSA.2009.5290800

applause (2015). https://github.com/applause/.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture*. Wiley.

Cloutier, R., Muller, G., Verma, D., Nilchiani, R., Hole, E., and Bone, M. (2010). The concept of reference architectures. *Systems Engineering*, 13(1):14–27.

Dageförde, J. C., Reischmann, T., Majchrzak, T. A., and Ernsting, J. (2016). Generating app product lines in a model-driven cross-platform development approach.

In *49th Hawaii International Conference on System Sciences (HICSS)*.

Evers, S., Ernsting, J., and Majchrzak, T. A. (2016). Towards a reference architecture for model-driven business apps. In *49th Hawaii International Conference on System Sciences (HICSS)*.

Fazal-e-Amin, Mahmood, A. K., and Oxley, A. (2011). An analysis of object oriented variability implementation mechanisms. *ACM SIGSOFT Software Engineering Notes*, 36(1):1.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Future of Software Engineering*, pages 37–54. IEEE Computer Society.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley.

Gartner Press Release (2015). http://www.gartner.com/newsroom/id/3169417.

Heitkötter, H., Hanschke, S., and Majchrzak, T. A. (2013a). Evaluating Cross-Platform Development Approaches for Mobile Applications. In *Revised Selected Papers WEBIST 2012*, volume 140 of *LNBIP*, pages 120–138. Springer.

Heitkötter, H., Majchrzak, T. A., and Kuchen, H. (2013b). Cross-Platform Model-Driven Development of Mobile Applications with MD^2. In *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 526–533. ACM.

Heitkötter, H., Majchrzak, T. A., and Kuchen, H. (2013c). MD^2-DSL - eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen. In *6. Arbeitstagung Programmiersprachen (ATPS)*, volume 215 of *LNI*, pages 91–106. Gesellschaft für Informatik e.V. (GI).

Heitkötter, H., Kuchen, H., and Majchrzak, T. A. (2015). Extending a Model-Driven Cross-Platform Development Approach for Business Apps. *Science of Computer Programming (SCP)*, 97(1):31–36.

Holzinger, A., Treitler, P., and Slany, W. (2012). Making apps useable on multiple different mobile platforms: On interoperability for business application development on smartphones. In *Multidisciplinary Research and Practice for Information Systems*, volume 7465 of *Lecture Notes in Computer Science*, pages 176–189. Springer.

J2ObjC (2015). http://j2objc.org/.

Joorabchi, M. E., Mesbah, A., and Kruchten, P. (2013). Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24.

Le Goaer, O. and Waltham, S. (2013). Yet another DSL for cross-platforms mobile development. In *Proc. of the First Workshop on the Globalization of Domain Specific Languages*, pages 28–33. ACM.

Majchrzak, T. A. and Ernsting, J. (2015). Reengineering an approach to model-driven development of business apps. In *8th SIGSAND/PLAIS EuroSymposium*, pages 15–31.

Majchrzak, T. A., Ernsting, J., and Kuchen, H. (2015). Achieving business practicability of model-driven cross-platform apps. *Open Journal of Information Systems (OJIS)*, 2(2):3–14.

map.apps product description (2015). http://conterra.de/en/produkte/con-terra-solutionplatform/mapapps/beschreibung.aspx.

Ohrt, J. and Turau, V. (2012). Cross-platform development tools for smartphone applications. *IEEE Computer*, 45(9):72–79.

Smith, J. (2009). Patterns – WPF Apps With The Model-View-ViewModel Design Pattern. https://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

Stahl, T. and Völter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

Swift Blog (2015). Swift blog - apple developer. https://developer.apple.com/swift/blog/.

WebRatio (2015). http://www.webratio.com/.