A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes

Christoph Rieger
ERCIS, University of Münster
christoph.rieger@uni-muenster.de

Herbert Kuchen
ERCIS, University of Münster
kuchen@uni-muenster.de

Abstract

App development has gained importance since the advent of smartphones to enable the ubiquitous access to information. Until now, multi- or cross-platform approaches are usually limited to different platforms for smartphones and tablets. With the recent trend towards app-enabled mobile devices, a plethora of heterogeneous devices such as smartwatches and smart TVs continues to emerge. For app developers, the situation resembles the early days of smartphones but worsened by the widely differing hardware, platform capabilities, and usage patterns. In order to tackle the identified challenges of app development beyond the boundaries of individual device classes, a systematic process built on the model-driven paradigm is presented. In addition, we demonstrate its applicability using the MAML framework to create interoperable business apps for both smartphones and smartwatches from a common, platform-independent model.

1. Introduction

With the rising importance of smartphones as pervasive mobile devices, small and task-oriented pieces of software called apps have emerged as new artifacts in software development. A major challenge of mobile apps lies in the diversity of devices and platforms, although ten years after the introduction of the iPhone, a duopoly of Android and iOS dominates the mobile operating system (OS) market. Recently, the market for *app-enabled* devices has become much more diverse as new device classes such as smartwatches and smart TVs become available to a broader audience. In the near future, further device classes – including smart personal assistants, smart glasses, and vehicles – are expected to reach the mainstream market. Within each device class, a multitude of devices exists with different hardware capabilities and platforms for which third-party apps can be provided.

As app-enablement does not automatically entail compatibility and portability, developing apps for these devices poses various challenges. The current situation resembles the early phase of exploration after the introduction of smartphones. Different approaches set out to bridge device-specific differences but development for heterogeneous app-enabled devices is a far more complex endeavor than creating apps for several platforms of a common device class. In addition, the term *cross-platform* as well as actual frameworks are usually limited to smartphones and sometimes – yet not always – technically similar tablets, ignoring the differing requirements and capabilities within the variety of other app-enabled devices. Therefore, an efficient method for developing truly cross-platform apps across the boundaries of individual device classes is required.

Focusing on the domain of *business apps* – i.e., form-based, data-driven apps interacting with back-end systems [1] –, we aim to simplify app development across multiple device classes by investigating three main research questions:

1. Which existing cross-platform approaches have the potential to be generalized in order to support a broad range of app-enabled devices?
2. How can a model-driven approach be structured to cater for peculiarities of multiple devices classes using platform-independent input models?
3. Is the domain-specific Münster App Modeling Language (MAML) applicable to different device classes using the proposed process model?

The remainder of this article follows these contributions. Section 2 highlights challenges arising from app development across device classes and analyzes current cross-platform approaches. Then, our proposal for a process model is presented in Section 3 and exemplified in Section 4 by extending the previously smartphone-centered MAML framework to create stand-alone apps for Wear OS smartwatches. Section 5 presents an evaluation of the presented model-driven approach and Section 6 discusses the applicability with regard to different scenarios. Finally, we revisit related work on app development combining different device classes in Section 7, before concluding in Section 8.

URI: https://hdl.handle.net/10125/60180
ISBN: 978-0-9981331-2-6
(CC BY-NC-ND 4.0)

2. Cross-platform app development

To extend the boundaries of current cross-platform development approaches, we broaden its scope to *app-enabled* devices, i.e., a device extensible with software that comes in small, interchangeable pieces which are usually provided by third parties unrelated to the hardware vendor or platform manufacturer and increase the versatility of the device after its introduction [2]. Although these devices are typically mobile or wearable and therefore related to the term *mobile computing*, there are further device classes with the ability to run apps (e.g., smart TVs). Because of the adaptation to heterogeneous device classes, new challenges arise, to which most approaches are not suited.

2.1. Challenges

Challenges related to app development across device classes can be grouped in four main categories [3].

Output heterogeneity Whereas most mobile apps are designed for screen sizes between 4" and 10", novel device classes vary greatly in terms of screen size (e.g., smartwatches $\leq$ 3" and smart TVs $\geq$ 20") or provide completely different means of output such as audio or projection. Even for screen-based devices, variability increases beyond "known" issues such as device orientation and pixel density: novel devices bring along completely different aspect ratios (e.g., ribbon-like fitness devices worn around the wrist) and form factors (e.g., round smartwatches) [2]. Therefore, the information output needs to be described on a high level of abstraction beyond a particular screen layout.

User input heterogeneity Correspondingly, novel app-enabled devices have different characteristics for user inputs which span from pushing hardware buttons to clicking on a graphical user interface (UI), tapping on touch screens, using auxiliary devices (e.g., stylus pens), and interacting via voice commands [2]. Moreover, multiple input alternatives may be available on one device and used depending on user preferences, usage context, or established interactions patterns of the respective platform. Again, this complexity calls for a higher level of abstraction that focuses on intended user actions rather than particular input events.

Device class capabilities Beyond user interfaces, hardware and software variability between different device classes are challenging. For example, the miniaturization in modern devices such as Wearables limits computational power and battery capacity. Complex computations may therefore be performed either on the device, offloaded to potential companion devices, or provided through edge/cloud computing [4]. Also, sensing capabilities can vary widely. Suitable replacements for unavailable sensors need to be provided, e.g., using manual map selection instead of GPS sensors.

Multi-device interaction Whereas cross-platform approaches often focus on providing the same functionality across devices of different users, additional complexity arises from multi-device interactions *per user*. This might occur *sequentially* when a user switches to a different device depending on the usage context or user preferences, e.g., using an app with smart glasses while walking and switching to the in-vehicle app when boarding a car. Alternatively, a *concurrent* usage of multiple devices for the same task is possible, for instance in second screening scenarios in which one device provides additional information or input/output capabilities [5]. In both cases, fast and reliable synchronization of content is essential.

2.2. Adequacy of existing approaches

A plethora of literature exists in the context of cross- or multi-platform development[1]. Classifications such as in [6] and [1] have identified five main approaches to multi-platform app development which are varyingly suited to the specific challenges of cross-platform development spanning different device classes.

With regard to runtime-based approaches, *mobile Web apps* – including recently proposed Progressive Web Apps – are mobile-optimized Web pages that are accessed using the device's browser and relatively easy to develop using Web technologies. However, most novel device types, e.g., the major smartwatch platforms WatchOS by Apple and Wear OS by Google (formerly Android Wear), do not provide Web views or browser engines that allow for HTML rendering and the execution of JavaScript code. This approach therefore cannot currently be used for targeting a broader range of devices.

Hybrid apps are developed similarly using Web technologies but are encapsulated in a wrapper component that allows accessing device hardware and OS functionality through an Application Programming Interface (API) and building app packages for installation via app marketplaces. As they rely on the same technology, hybrid apps can neither be used beyond smartphones and tablets.

In contrast, apps using a *self-contained runtime* do not depend on the device's browser engine but use platform-specific libraries to use native UI components. Of the runtime environment approaches, this is the only one that can be used for developing truly cross-platform apps. Although usually based on custom

[1] We use these terms synonymously in this paper, although "multi-platform" appears more often in papers combining Web and mobile platforms.

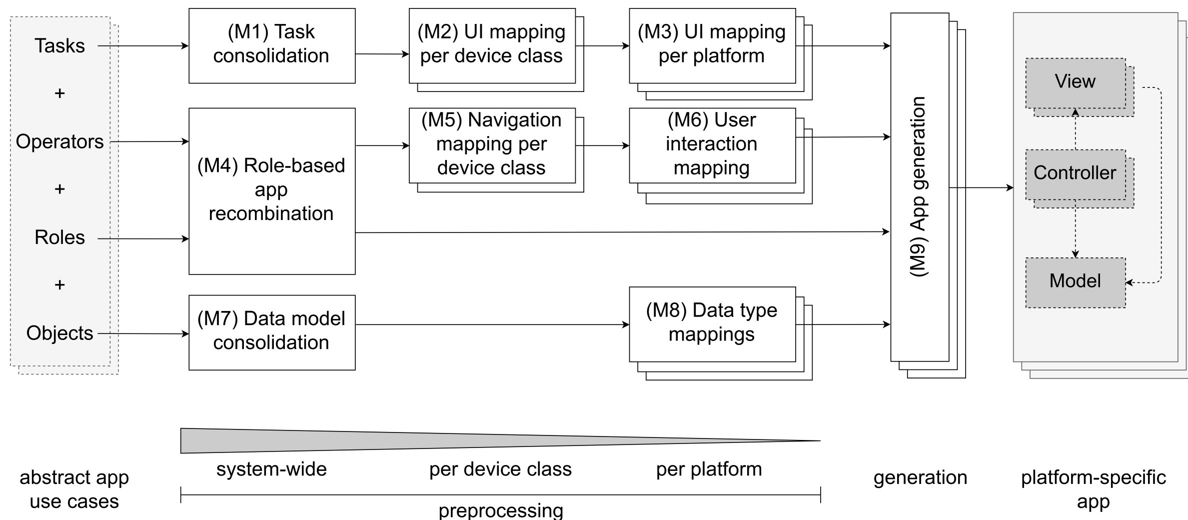

Figure 1. Process model for cross device class app development

scripting languages, a runtime can also be used as a replacement for inexistent platform functionality. As an example, CocoonJS[2] recreated a restricted Webview engine and therefore supports the development of JavaScript-based apps also for Android and iOS. However, synergies with regard to user input/output and available hardware/software functionality across heterogeneous devices are limited by the runtime's API.

Considering generative approaches to cross-platform development, *model-driven software development* has several advantages as it uses textual or graphical models as main artifacts to develop apps and then generates native source code from this platform-neutral specification. Modeling custom domain-specific concepts allows for a high level of abstraction, for example circumventing issues such as input and output heterogeneity using declarative notations. Arbitrary platforms can be supported by developing respective generators which implement a suitable mapping from descriptive models to native platform-specific implementations.

Finally, *transpiling* approaches use existing application code and transform it to different programming languages. Bridging device classes using this approach is technically feasible as the result is also native code. However, there is more to app development than just the technical equivalence of code, which also explains the low adoption of this approach by current cross-platform frameworks. For instance, user interfaces behave drastically differently across different device classes, and substantial transformations would be required. It is therefore unlikely that this approach is viable for bridging device classes beyond reusing individual components such as business logic.

To sum up, only self-contained runtimes and model-driven approaches are candidates for device class spanning app development, of which the latter additionally benefits from the transformation of domain abstractions to platform-specific implementations.

3. A process model for app development across device classes

Although not all types of apps make sense on all types of devices, our focus lies on *business apps* whose workflow-like concepts of subsequent data manipulation activities can be transferred to the particular user experiences on heterogeneous devices. However, the complexity of developing apps across device classes requires a structured approach in order to manage the variability of hardware and software capabilities. Based on the applicability of current cross-platform development techniques, we propose an extended model-driven approach to develop business apps as depicted in Figure 1.

In order to structure the resulting application and foster comprehensibility for all development stakeholders, the domain-specific notation should be designed with modularity in mind and allow to specify different *use cases*, i.e., units of functionality comprising a self-contained set of behaviors and interactions performed by the app user [7]. In a workflow context, a use case can also be interpreted from a user task model perspective [8]. The semantically compliant ConcurTaskTree notation for example consists of four elements [9]:

[2]https://docs.cocoon.io/article/canvas-engine/

- the abstract and descriptive *task* descriptions which together form the use case's functionality,
- *operator*s defining the allowed sequences of executed tasks, representing an abstract notion of navigation actions within a use case
- the user *role*s that are allowed to interact with the system (and might differ per task), and
- the data *objects* to interact with in each task.

From such a descriptive representation, mappings need to be defined in order to reach platform-adapted app output. To handle the transformation complexity, we propose a step-wise refinement of input models. First, system-wide transformations need to be applied to consolidate multiple modular and independently modeled use cases (M1, M4, M7). Second, the model is preprocessed according to each targeted device class (M2, M5). The extent of these transformations highly depends on the degree of maturity of the respective device class. For example, common layout patterns such as tabs or vertical content scrolling may be observed frequently and smartphone hardware converges with regard to sensors and screen dimensions. Third, platform-specific preprocessing is required, similar to today's model-driven cross-platform approaches (M3, M6, M8). Only in this step, a final mapping of abstract UI elements to concrete widgets based on a choice of possible representations for the given task can be specified. Finally, the code generation is executed which outputs the platform-specific app artifacts (M9). The Model-View-Controller (MVC) pattern or derived patterns such as Model-View-ViewModel (MVVM) are suited as high-level structure of the resulting applications [10]. In addition, reference architectures applicable to platforms of the same device class can be used to maintain consistency in this final step, thus easing the development of new generators and the addition of features across existing platform generators [11].

Using the proposed model, the challenges of app development spanning device classes can be tackled:

Resolving output heterogeneity Based on the descriptive use case models with a high level of abstraction, the main activity related to task consolidation (M1) is the specification of views from the logical task units of functionality. If required, generic model preprocessing and simplification activities are also performed, e.g., the resolution of shorthand definitions and references within the models. Subsequently, mapping the UI per device class (M2) is achieved by applying two types of transformations that do not modify the content but specify a target layout for the views adapted to the device class: On the one hand, the presentation of information can be *layouted* according to the available screen sizes, for example by choosing an appropriate appearance – one could think of tabular vs. graphical representations – or leave out complementary details, if necessary. On the other hand, the content can be *re-formatted* by an adaptive UI according to usual interaction patterns, e.g., to present large amounts of information through scrolling, subdivided into multiple subsequent steps, or as a hierarchical structure [12].

Finally, a platform-specific UI mapping (M3) defines concrete UI elements and the actual user interactions within a view. Activities within this mapping include the selection of suitable widgets, their arrangement within the previously specified layout container as well as bidirectional data bindings and validation.

To exemplify the difference to the previous device class preprocessing, an "item selection" task within a use case might for instance be mapped to an abstract list selection for all tablet platforms. The concrete representation such as a horizontal card-based layout for Windows or a vertical list view for iOS is then specified in the platform-specific mapping stage.

Resolving input heterogeneity User inputs are not only important for entering data but also to navigate within the app. The large variety of device class specific (e.g., remote controls for TVs), platform-specific (e.g., Android's hardware buttons), and even device-specific input events (e.g., Apple's 3D touch gestures) is a hurdle for efficient modeling. Instead, user inputs should be described as *intended actions* for completing a particular task. Based on the possible sequences of tasks, navigation paths can be established (M5), including the initial task selection when opening the app, back-and-forth navigation between views, and conditional process flows resulting from user decisions.

In the platform-specific mapping (M6), it is then possible to perform a mapping of actions to actual input mechanisms. This is similar to the decoupling mechanism for user inputs in MVC architectures described in [13] but on a higher level of abstraction. For instance, a "back" action can be linked to a hardware button, displayed in a navigation bar on screen, bound to a right-swipe gesture in Wear OS, or recognized by a spoken keyword. Some novel device classes such as smart cars are in early experimental stage; mappings such as specific gestures then represent preliminary design decisions suitable for the range of possible apps and in accordance with vendor guidelines. As noted above, repetitive platform-specific transformations can of course be shifted to a more generic layer in the future when commonalities become apparent.

Managing device capabilities Whereas tasks, operators, and roles have no interdependencies between use cases, a global data layer needs to be established from different data models (M7). Data model inference

can validate the compliance of different use cases and also provide additional modeling support for the editing tools of the domain-specific language (DSL) [14]. Subsequently, non-primitive data types (e.g., dates and colors) need to be mapped to available platform concepts for output to the user and back-end communication (M8).

In the context of business apps, there are usually no complex computation tasks to be performed on the device itself. Derived attribute values may be computed at runtime but if necessary, computations may be offloaded to remote computation providers. With regard to sensors, a progression in functionality should be aimed for: the same app would function on many devices while providing the highest level of functionality achievable on the given hardware. For instance, location information is easily retrieved if a GPS sensor is available but generic approaches need to cater for adequate fall-back mechanisms such as manual selection of addresses on a map or address lookup.

Enabling multi-device interaction In multi-user scenarios, a recombination of tasks needs to be performed to account for distinct user roles (M4). This mapping modifies the sequence of tasks to cater for the interruption of activities and the automatic transmission of application state to the subsequent role. As a result of this decomposition and bundling of use cases, either one app is created that supports all user roles (depending on the logged-in user) or different apps are generated per role.

In addition, synchronization is essential both for the sequential and concurrent usage of apps on multiple heterogeneous devices. In contrast to "traditional" process-oriented apps, information not only needs to be propagated between devices after a given task has been completed but also intermediate states of data or even a live synchronization capability including incomplete user inputs (e.g., per character) is required. Also, the workflow state itself must be captured in order to pass the current process instance to other concurrently used devices as well as to different users when role changes occur. Consequently, the back-end component also needs to manage the relationship of users and devices (either via central device registration or tracking on which device the user is currently logged in) and provide push-based or pull-based update mechanisms.

4. Realizing cross device class apps

In this section, the open-source MAML framework[3] is briefly introduced to demonstrate the applicability of the proposed process. This domain-specific notation was chosen because of its fit for business apps and its high-level abstraction [14, 15].

[3]MAML code repository: https://github.com/wwu-pi/maml

4.1. The MAML framework

MAML is a graphical DSL for specifying business apps and based on five main design goals [15]:

- *Automatic cross-platform app creation* by transforming a graphical model to fully functional source code for multiple platforms.
- *Domain expert focus* to allow non-technical stakeholders to create, alter, or communicate about an app using the actual models.
- *Data-driven process modeling* specifies the application domain on a high level of abstraction by interpreting data manipulation as a process.
- *Modularization* of activities in distinct use cases helps for maintenance, e.g., for domain experts.
- *Declarative description* of the complete app, including necessary specifications of data model, business logic, user interactions, and UI views.

We illustrate the notation and applicability of MAML for cross device class app development using the common scenario of a to-do management app. The system can be represented with two *use cases* depicted in the screenshots of the implemented drag&drop editor component depicted in Figure 2. In the first use case, one or more tasks are created with corresponding attributes such as due date and responsibility assignment. The second use case lists all current to-dos and upon selection of an item shows editable details and the possibility to complete the task. The sequence of process steps is modeled between a *start event* (labeled with (a) in Figure 2) and *end events* (b). A *data source* (c) specifies the main type of the manipulated entity which is either stored locally or on a remote back-end system. Subsequently, *interaction process elements* (d) are used to *create/show/update/delete* an entity, but also to *display messages* or access device sensors. Note that this strictly declarative description does not make assumptions on the appearance on a device. To perform steps without user interaction, *automated process elements* (e) can be used to *transform* data values or navigate between objects, request information from *web services*, or *include* existing use cases for model reuse. *Process connectors* signify the order of process steps, represented by a default "Continue" action unless specified differently (f). *Condition*s branch out the process flow based on a manual user action (using differently labeled connectors; (g)), or automatically by evaluating expressions referring to the considered object (not visualized in the example).

In addition, the use case models contain the data linked to each process step. *Attributes* (h) are modeled as combination of a name, the data type, and the respective cardinality. Several data types such as *String, Integer, Float, PhoneNumber, Location* etc. are

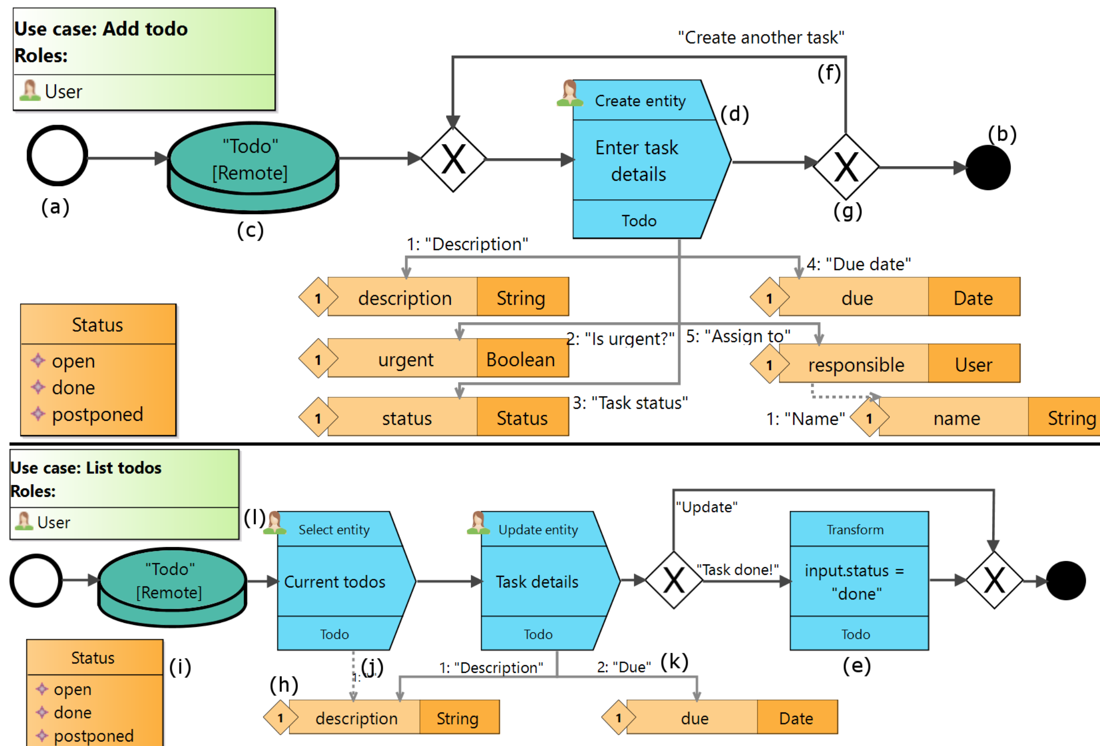

Figure 2. Sample MAML models for a to-do list use case

already provided but the modeler can define additional custom types which are further described using nested attributes (e.g., the "User" type in Figure 2 specifies a "name" attribute). Moreover, the modeler can create *enumeration* types (i) with lists of possible values. A suitable UI representation is automatically chosen based on the *parameter connector*: Dotted arrows (j) signify a reading relationship whereas solid arrows (k) represent a modifying relationship. This refers not only to the manifest representation of attributes displayed either as read-only text or editable input field. The interpretation also applies in an abstract sense, e.g., regarding web services which *read* input parameters and *modify* information through their response. Each connector also specifies an order of appearance and a human-readable caption.

Finally, annotating freely definable *roles* (l) to process elements allows modeling processes with several persons involved, e.g., in scenarios such as approval workflows. When a role change occurs, the app automatically informs eligible users about the open process instance.

4.2. Business apps for smartphone and smartwatch with MAML

As proof of concept for this work, the MAML framework (which supports app generation for Android and iOS smartphones from the same input models [15]), was extended by a smartwatch generator for Wear OS and prepared for further app-enabled device classes. Although the utilized DSL is designed to be platform-independent, existing generators focused on transformations towards smartphone apps. Therefore, the process model described in the previous section was applied to structure and separate the growing generation capabilities – using the BXtend framework and Xtend language for implementing the model transformations [16]. Because of space constraints, the respective transformations are only sketched next.

Regarding *system-wide preprocessing*, separate MAML input models are recombined into a single app project. The task consolidation transformation (M1) merges multiple use cases into a single app project and prepares required app-internal components, e.g., web service calls. A role-based app recombination mapping (M4) separates each use case into multiple process sequences which are performed by the same role in order to generate distinct apps. Because no explicit data modeling is required in MAML, a common data model (M7) needs to be inferred both within use cases and for the overall app project (cf. [14]). The output and all further preprocessing is performed based on a second, textual, DSL called MD^2 [1] with more detailed specifications, especially with regard to the view layer.

Figure 3. Screenshots of the resulting Wear OS smartwatch app

Subsequently, *device class specific* preprocessing is performed by repartitioning views to fit the amount of fields to the available screen sizes (M2). For example, to-do elements can be created within a single view on a smartphone, but the smartwatch implementation splits this in two subsequent steps to avoid scrolling. Also, suitable layouts (e.g., tab or scroll-based) are selected based on the modeled task type – still using abstract UI elements. Regarding the navigation mapping (M5), an additional start-up view is included for all smartphone apps to let the user choose a startable use case when entering the app.

Finally, concrete widgets are introduced in the *platform-specific preprocessing* of UI elements (M3), for instance exploiting the round design of a smartwatch by a curved list widget. Default actions are transformed to prominently displayed buttons in the navigation bar in iOS, whereas Wear OS uses so-called ActionDrawers[4], e.g., to trigger the "Add todo" use case ("+" icon in Figure 3) while reading through the list of to-dos. In the user interaction mapping (M6), the navigation through the process is mapped to buttons or specific patterns such as the NavigationDrawer[4] in Wear OS.

During the generation phase (M9) of MAML-based apps, an MVC separation of concerns is established with data bindings to view elements [11] which results in structured apps that may be customized with individual code. For the sample app model represented in Figure 2, resulting app screenshots of the generated smartwatch and smartphone apps are depicted in Figures 3 and 4.

[4] https://designguidelines.withgoogle.com/wearos/components/

5. Evaluation

To evaluate MAML as common notation to develop both smartphone and smartwatch apps, a study was performed with 23 Master students attending a course on model-driven software development. Their previous experience with app development was limited, with an average response value of 3.26 regarding Web apps (on a 5-level Likert scale from expert to little experience) as well as hybrid and native apps with 4.35 and 4.3, respectively. After presenting the to-do app scenario depicted in Figure 2 together with a short introduction to MAML, the participants were asked to sketch a suitable smartwatch application. 83% of them imagined a scrollable list to display to-dos (65% vertically, 17% horizontally), and 30% added an action button to create new to-dos from there. Of those participants who visualized view transitions, accessing task details was mostly considered by tapping on the list element on screen (42%), other participants opted for dedicated buttons (33%), hardware buttons (8%), swipe gestures (8%), and speech interfaces (84%).

Regarding the representation of task details/creation forms, participants modeled different view designs such as a single, scrollable view (35%), individual steps for each attribute (30%), a separation into multiple views according to available space (9%), or voice input (30%). Again, navigating back and forth in the dialog was envisaged either using on-screen/physical buttons (35% / 8%) or swipe gestures (22%). This diversity of interfaces reflects the variety of interaction modes exhibited by today's smartwatches.

Subsequently, the generated apps (Figures 3 and 4) were presented to the participants to compare the implementation with their expectations. The participants agreed that the generated interface suitably represents the modeled process (2.04) and that the app is functional (2.39). The moderated visual appeal (3.30) is a drawback resulting from the generic nature of business apps our generator has to cater for. However, the participants agreed that using model-driven transformations on a single input model drastically accelerates the development speed (2.48), estimating an average drop from 27 hours of manually programming a mobile application to 41 minutes for creating the equivalent MAML models. Overall, the participants of this preliminary study did not perceive particular complexities resulting from the implicit specification of multiple apps with MAML (3.35) and rated the approach suitable for creating apps across device classes (2.55).

Furthermore, a 10-question System Usability Scale (SUS) questionnaire was filled out to calculate a usability score (on a [0;100] interval) [17]. Compared to an earlier

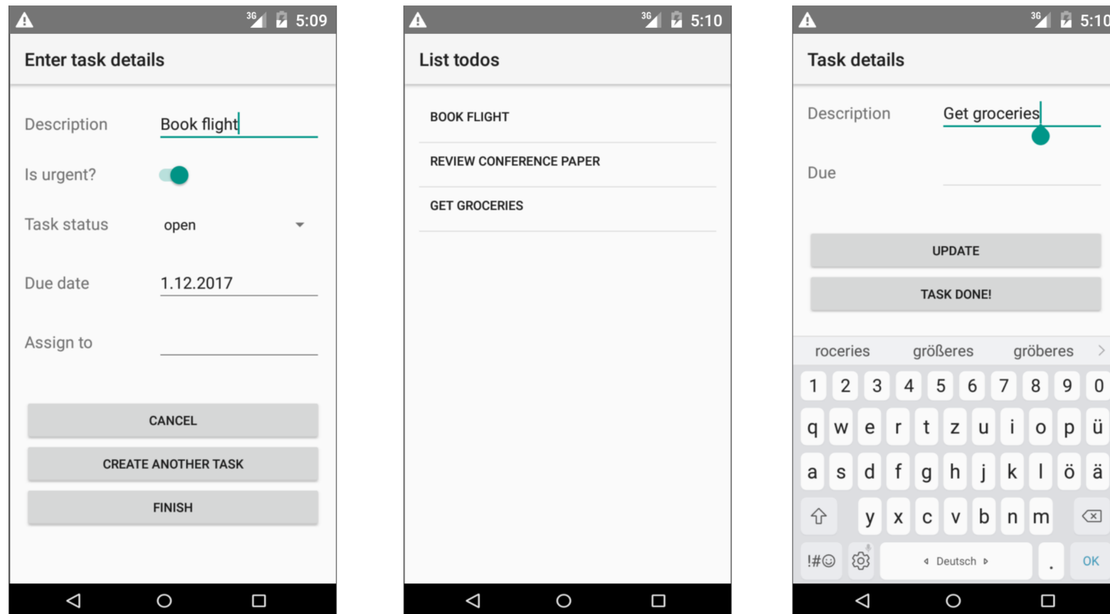

Figure 4. Screenshots of the resulting Android smartphone app

evaluation of MAML for smartphone app generation reaching a score of 66.83 [15], similar results could now be obtained with regard to smartwatch with a 66.85 score ($\sigma = 12.95$). The perceived usability of MAML therefore barely depends on the tested devices which underlines the objective of a platform-agnostic representation. Using the proposed process model, the vision of model-driven app development for a broad range of devices can be put into practice while avoiding the time-intensive repetitive adaptation of high-level models during generator implementation.

6. Discussion

The process model presented in Figure 1 does not aim for an unchangeable, all-encompassing set of transformation rules but describes the required transformation steps broken down into individual activities in a structured manner. Likewise, the approach supports a wide range of technical implementations regarding input and output of transformations which can be realized using appropriate technologies. In this regard, our approach differs from the Model-Driven Architecture (MDA) by not enforcing a strict assignment of *platform-independent* and *platform-specific* models to the respective steps. A platform-specific intermediate model might be omitted completely if the final preprocessing is executed within the generator component. Moreover, depending on the design of the chosen DSL, all transformations might be performed according to a single meta model or using multiple representations with different granularity. In the example of MAML as abstract representation, the subsequent transformations are implemented using another DSL, e.g., providing more detailed view specifications. The commonly cited Cameleon Reference Framework (CRF) [18] similarly describes a step-wise refinement from task-oriented specification to abstract, concrete, and final UI elements. However, it focuses solely on the interface aspect (i.e., the top row in Figure 1) as compared to the additional perspectives on navigation, business logic, and data layer.

The main stakeholders of the approach include not only software engineers and developers in the field of mobile apps (working mainly on the transformations) but also process modelers and non-technical users (capturing requirements and providing the abstract representation). Additional stakeholders are integrated in specific transformations as needed, for example requesting design expertise for the platform-adapted UI mapping, and human-computer interaction experts for navigating through the resulting apps. The responsibility of maintaining the transformation chain can be assigned to teams focused either on individual refinement steps of the preprocessing (vertical subdivision) or according to a functional domain (horizontal subdivision).

It is clear that a particular app does not need to make sense on all possible target platforms but the additional effort of generating apps for all devices at once using the same input models is marginal. In return, the potential of upcoming app-enabled devices is tapped and may provide opportunities for enhancing productivity as illustrated with three exemplary use cases in the following.

1) Dashboard apps: Many teams track key performance indicators for monitoring progress, motivating team members, and broadcasting daily achievements. Besides entering this information via personal devices, dashboards can be displayed on large screens in the office. Heterogeneous team member equipment is a known challenge in cross-platform development, but now intensified because of new devices for visualization (smart TV) or notifications (Wearables).

2) Logistics apps: Order picking in warehouses is an activity that is mainly coordinated using information systems. Operating hands-free using augmented reality (AR) devices instead of traditional scanners significantly improves the productivity of employees and first commercial products for the logistics industry already exist[5]. Similarly, technicians can benefit from up-to-date information both before and during repair works.

3) Field service apps: Salespersons need flexible access to their company's information systems from different contexts, e.g., while traveling or in sales talks with customers. The freedom to choose from multiple owned devices (e.g., smartphone, tablet, or smartwatch) for communication, notifications, or information access helps them to accomplish tasks more efficiently.

A limitation of the integrated approach from abstract specification to source code lies in the tight coupling of the app development process for various platforms, unless work can be coordinated among different teams as mentioned above. In addition, the allocation of specific mapping activities to the presented phases of the process model is final but a continuous process. Depending on the heterogeneity or convergence within a device class over time, common platform characteristics may be shifted to earlier mappings in the process chain. As the proposed process model is not limited to a specific technology, further instantiations are desirable to validate the approach. Regarding limitations of the the newly created Wear OS generator, the implementation is still in a prototypical state and we are working on improving the UI/UX design of its output. However, it is integrated in the MAML framework and already contributes functional apps for the new platform target.

7. Related work

Whereas a plethora of literature exists in the context of cross-/multi-platform development, previous work on app development *spanning multiple device classes* is much more scarce. Although an extensive literature search was performed, not a single paper out of almost 300 initial results provided a comprehensive theory on app development across device classes. This shows that app development beyond smartphones is not yet approached systematically but on a case-by-case basis.

Cross-platform overview papers such as [19] typically focus on a single category of devices and apply a very narrow notion of mobile devices, e.g., when Humayoun et al. [20] examine "the diversity in smart-devices (i.e., smartphones and tablets)". [2] provides the only classification beyond those "classical" mobile devices. To complicate matters, some papers mention that there are novel devices as visionary outlook but do not detail on actual app development challenges ([21, 22]). In contrast to "liquid software" aiming for portability across heterogeneous platforms [23], model-driven approaches can integrate with arbitrary infrastructures.

Only few papers provide a technical perspective on novel app-enabled devices: Singh and Buford [24] describe a cross-device team communication use case for desktop, smartphones, and Wearables, and Esakia et al. [25] performed research on Pebble smartwatch and smartphone interaction in computer science courses. Some authors consider specific combinations of devices, for example Neate et al. [5] who analyze second screening apps that combine smart TVs with additional content on smartphones or tablets, and Koren and Klamma [26] who propose a middleware approach to integrate data and UI of heterogeneous Web of Things devices. Zhang et al. [27] present an architecture for the future of "transparent computing" using virtual apps on lightweight devices equipped with UEFI firmware interface but do not focus on the apps themselves. Finally, a few papers on mobile (cloud) computing might be applicable to app development across device classes but do not explicitly mention this potential.

With regard to commercial cross-platform products, Xamarin[6] and CocoonJS[2] are two runtime-based approaches that provide Wear OS support to some extent. Whereas several other frameworks claim to support Wearables, this usually only refers to data access or the display of notifications by the main smartphone application on coupled devices. Only the domain of gaming apps exhibits more diversity. Unity3D[7], an engine for 2D/3D games, supports 29 platforms including smartphones, smart TVs, consoles, and AR devices.

While related to research fields such as self-adaptive UI or model-driven UI development [28], our work concentrates on the automated generation of all app perspectives for a specific platform without need for dynamic UI adaptation at runtime. Context-awareness and adaptation (e.g., to screen sizes) are therefore complementary considerations when designing mappings for a device class or specific platform.

[5] http://www.smartpick.be/

[6] https://developer.xamarin.com

[7] https://unity3d.com

8. Conclusion and outlook

The field of modern mobile computing does not show signs of less rapid progress and novel app-enabled devices are constantly emerging. New challenges resulting from the heterogeneity of input and output mechanisms, device class capabilities, and multi-device interaction require consideration by app developers. In this article, we have analyzed existing cross-platform development approaches for efficiently creating apps for heterogeneous devices. We propose a systematic process model based on the model-driven paradigm which supports the inclusion of novel app-enabled devices. The applicability of this process is demonstrated using the MAML framework that utilizes a graphical DSL for generating business apps for smartphones and now also smartwatches. Because of its high level of abstraction, the challenges of app development across device classes can be tackled through a multi-step preprocessing of input models towards platform-specific code generation.

Although the smartwatch generator for MAML demonstrates the required concepts, real-world adoption is pending. Future work concentrates on generating apps for other platforms to validate the generalizability of the proposed model, especially for devices without touchscreen such as smart personal assistants.

References

[1] T. A. Majchrzak, J. Ernsting, and H. Kuchen, "Achieving business practicability of model-driven cross-platform apps," *OJIS*, vol. 2, no. 2, pp. 3–14, 2015.

[2] C. Rieger and T. A. Majchrzak, "Conquering the mobile device jungle: Towards a taxonomy for app-enabled devices," in *WEBIST*, pp. 332–339, 2017.

[3] C. Rieger and H. Kuchen, "Towards model-driven business apps for wearables," in *Mobile Web and Intelligent Information Systems* (M. Younas, I. Awan, G. Ghinea, and M. Catalan Cid, eds.), pp. 3–17, Springer, 2018.

[4] A. Reiter and T. Zefferer, "Power: A cloud-based mobile augmentation approach for web- and cross-platform applications," in *CloudNet*, pp. 226–231, IEEE, 2015.

[5] T. Neate, M. Jones, and M. Evans, "Cross-device media: A review of second screening and multi-device television," *Personal and Ubiquitous Computing*, vol. 21, no. 2, pp. 391–405, 2017.

[6] W. S. El-Kassas, B. A. Abdullah, A. H. Yousef, and A. M. Wahba, "Taxonomy of cross-platform mobile applications development approaches," *Ain Shams Engineering Journal*, 2015.

[7] Object Management Group, "Unified modeling language 2.5," 2015.

[8] D. Sinnig, P. Chalin, and F. Khendek, "Common semantics for use cases and task models," in *Integrated Formal Methods* (J. Davies and J. Gibbons, eds.), vol. 4591 of *LNCS*, pp. 579–598, Springer, 2007.

[9] F. Paternò, *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000.

[10] A. Syromiatnikov and D. Weyns, "A journey through the land of model-view-design patterns," in *WICSA*, pp. 21–30, 2014.

[11] J. Ernsting, C. Rieger, F. Wrede, and T. A. Majchrzak, "Refining a reference architecture for model-driven business apps," *WEBIST*, pp. 307–316, 2016.

[12] J. Eisenstein, J. Vanderdonckt, and A. Puerta, "Applying model-based techniques to the development of UIs for mobile computers," *IUI*, 2001.

[13] A. Carcangiu, G. Fenu, and L. D. Spano, "A design pattern for multimodal and multidevice user interfaces," in *EICS*, pp. 177–182, ACM, 2016.

[14] C. Rieger and H. Kuchen, "A process-oriented modeling approach for graphical development of mobile business apps," *COMLAN*, vol. 53, pp. 43–58, 2018.

[15] C. Rieger, "Evaluating a graphical model-driven approach to codeless business app development," in *HICSS*, pp. 5725–5734, 2018.

[16] T. Buchmann, "Bxtend - a framework for (bidirectional) incremental model transformations," in *MODELSWARD*, 2018.

[17] J. Brooke, "Sus-a quick and dirty usability scale," in *Usability evaluation in industry*, pp. 189–194, 1996.

[18] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, Q. Limbourg, L. Marucci, F. Paterno, C. Santoro, N. Souchon, D. Thevenin, *et al.*, "The CAMELEON reference framework," *D1.1*, 2002.

[19] C. Jesdabodi and W. Maalej, "Understanding usage states on mobile devices," in *Int. Joint Conf. on Pervasive and Ubiquitous Computing*, pp. 1221–1225, ACM, 2015.

[20] S. R. Humayoun, S. Ehrhart, and A. Ebert, "Developing mobile apps using cross-platform frameworks: A case study," in *HCI International*, pp. 371–380, 2013.

[21] E. Umuhoza, "Domain-specific modeling and code generation for cross-platform multi-device mobile apps," *CEUR Proceedings*, vol. 1499, 2015.

[22] J. C. Dageförde, T. Reischmann, T. A. Majchrzak, and J. Ernsting, "Generating app product lines in a model-driven cross-platform development approach," in *HICSS*, pp. 5803–5812, 2016.

[23] A. Gallidabino, C. Pautasso, V. Ilvonen, T. Mikkonen, K. Systä, J. P. Voutilainen, and A. Taivalsaari, "On the architecture of liquid software: Technology alternatives and design space," in *WICSA*, pp. 122–127, April 2016.

[24] K. Singh and J. Buford, "Developing WebRTC-based team apps with a cross-platform mobile framework," *IEEE CCNC*, 2016.

[25] A. Esakia, S. Niu, and D. S. McCrickard, "Augmenting undergraduate computer science education with programmable smartwatches," in *SIGCSE*, pp. 66–71, 2015.

[26] I. Koren and R. Klamma, "The direwolf inside you: End user development for heterogeneous web of things appliances," *ICWE*, pp. 484–491, 2016.

[27] Y. Zhang, K. Guo, J. Ren, Y. Zhou, J. Wang, and J. Chen, "Transparent computing: A promising network computing paradigm," *CISE*, vol. 19, no. 1, pp. 7–20, 2017.

[28] P. A. Akiki, A. K. Bandara, and Y. Yu, "Engineering adaptive model-driven user interfaces," *IEEE Transactions on Software Engineering*, vol. 42, pp. 1118–1147, Dec 2016.