Towards Model-Driven Business Apps for Wearables

Christoph Rieger(✉) and Herbert Kuchen

ERCIS, University of Münster, Münster, Germany
{christoph.rieger,kuchen}@uni-muenster.de

Abstract. With the rise of wearable devices expected to continue in the near future, traditional approaches of manually developing apps from scratch for each platform reach their limits. On the other hand, current cross-platform approaches are usually limited to platforms for smartphones and tablets. The model-driven paradigm seems well suited for developing apps for novel and heterogeneous devices. However, one of the main challenges for establishing a model-driven framework for wearables consists of bridging the variety of user interfaces and considering different capabilities of device input and output. This paper seeks to investigate the challenges of app development for wearable devices regarding user interfaces and discusses a possible mapping of typical application building blocks in the domain of business apps. Ultimately, apps modelled on a task-oriented level of abstraction using platform-independent notations such as MAML or CTT can then be transformed into code that adopts device class specific representations.

Keywords: Cross-platform · Model-driven software development
Multi-platform · Wearables · Mobile app · Business app

1 Introduction

Wearables have seen a drastic increase in popularity in the last years, with most major vendors providing software platforms and hardware products such as smartwatches and fitness devices for the mainstream consumer market. According to Gartner, the wearable device market is expected to grow significantly in the forthcoming years to more than 500 million devices sold in 2021 [23]. Many of these devices will be app-enabled and accessible to third-party developers.

For now, many available wearable applications – so-called apps – are of rather exploratory nature, providing only a limited set of companion functionality compared to a main app running on a smartphone. However, traditional approaches to app development are limited with regard to the plethora of input and output capabilities by current and announced wearable devices such as smartwatches and smart glasses. In addition, not all cross-platform approaches can be used for developing apps for wearables due to technical limitations. Many devices do not provide web views or support JavaScript execution, excluding hybrid frameworks such as the popular Apache Cordova.

© Springer International Publishing AG, part of Springer Nature 2018
M. Younas et al. (Eds.): MobiWIS 2018, LNCS 10995, pp. 3–17, 2018.
https://doi.org/10.1007/978-3-319-97163-6_1

Therefore, new methods are needed in order to extend app development to these different device classes. Model-driven software development (MDSD) seems particularly suitable for targeting a large range of devices and ease the development as opposed to the repetitive manual implementation. When extending existing model-driven mobile development approaches to wearables, adapting business logic is probably not the most pressing issue and transpiling approaches such as ICPMD [6] may be applied. One of the main challenges consists of bridging the heterogeneity of user interfaces (UI) regarding input capabilities and information visualisation according to the platform's user experience (UX).

In addition, wearable devices are not used in isolation but often considered as connected devices – although they might technically run standalone apps such as with Google's Wear OS (formerly Android Wear 2.0). Most likely, wearable apps in the near future will co-exist with other device counterparts or interact with their environment such as with smart personal agents or cyber-physical sensor networks. With regard to these scenarios, MDSD can play out its strengths even more when apps are jointly modelled for multiple devices using a single, abstract representation, and individually transforming the models to device-specific interfaces.

This paper seeks to investigate the challenges of app development for wearable devices, in particular related to the diversity issue of device interfaces. As a building block on the path towards extending model-driven development approaches to wearable devices, it contributes a conceptual mapping of UI representations across multiple device classes for typical operations in the domain of business apps. The eventual aim is to spark discussion on the long-standing issues of cross-platform UI modelling. The structure of this paper follows these contributions: after discussing related work in Sect. 2, we highlight the current challenges and our proposed mapping of user interfaces and suitable modelling approaches in Sect. 3. Section 4 discusses the applicability of model-driven development for wearable devices before we conclude in Sect. 5.

2 Related Work

In this work, we focus on the domain of business apps, i.e. form-based, data-driven applications interacting with back-end systems [13]. Using this definition, business apps not only refer to smartphone applications but also apply to a broader scope of *app-enabled* devices. These can be described as being extensible with software that comes in small, interchangeable pieces that are usually provided by third parties unrelated to the hardware vendor or platform manufacturer and increase the versatility of the device after its introduction [19]. Although related to the term *mobile computing*, app-enablement also considers stationary devices such as smart TVs, smart personal assistants, or smart home devices.

Cross-platform overview papers such as [11] typically focus on a single category of devices and apply a very narrow notion of mobile devices. [19] provides the only classification that includes novel device classes. Furthermore, few papers provide a technical perspective on apps *spanning multiple device classes*. Singh and Buford [21] describe a cross-device team communication use case for desktop, smartphones, and wearables, and Esakia et al. [8] performed research on Pebble smartwatch and smartphone interaction in computer science courses. In the context of Web-of-Things devices, Koren and Klamma [12] propose a middleware approach to integrate data and heterogeneous UI, and Alulema et al. [1] propose a DSL for bridging the presentation layer of heterogeneous devices in combination with web services for incorporating business logic.

With regard to commercial cross-platform products, Xamarin[1] and CocoonJS[2] provide Android Wear support to some extent. Whereas several other frameworks claim to support wearables, this usually only refers to accessing its data by the main smartphone application or displaying notifications on coupled devices.

Together with the increase in devices, new software platforms have appeared, some of which are either related to established operating systems for other device classes or are newly designed to run on multiple heterogeneous devices. Examples include Wear OS, watchOS, and Tizen. Although these platforms ease the development of apps (e.g., reusing code and libraries), subtle differences exist in the available functionality and general cross-platform challenges remain.

3 Creating Business App UIs for Wearable Devices

To pave the way towards model-driven software development for wearable devices, we provide a possible mapping of typical tasks in this domain to different app-enabled devices and present a model-driven app development approach.

3.1 Challenges of Wearable UIs

From development and usage perspectives, two main categories of UI/UX challenges related to app development across different device classes can be identified.

Diversity of Input and Output Capabilities. Traditionally, mobile apps are designed for rectangular screen sizes between 4" and 10" to cover smartphones and tablets with similar visual characteristics. However, wearables vary greatly in terms of screen size or provide completely different means of output such as audio or projection. In addition, current interface design considerations such as device orientation and pixel density are aggravated due to the introduction of different aspect ratios (e.g. ribbon-like fitness devices worn around the wrist), positions (e.g. objects at different angles and depths within the field of view for smart glasses) or form factors (e.g. round smartwatches) [19].

[1] https://www.xamarin.com.
[2] https://docs.cocoon.io/article/canvas-engine/.

Correspondingly, novel app-enabled devices provide different possibilities for user interaction which span from hardware buttons to handling graphical UI elements on touch screens, using auxiliary devices (e.g., stylus pens), and voice inputs [19]. Moreover, multiple input alternatives may be available on one device and used depending on user preferences or usage context.

Multi-device Usage Patterns. Until now, cross-platform approaches were mainly designed to provide equivalent functionality for similar devices with different operating systems. However, novel app-enabled devices usually do not replace smartphone usage but represent complementary devices which are used contextually (e.g. location- or time-based) or depending on user preferences. This might occur *sequentially* when a user switches to a different device, 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 for stationary devices in the room [14]. Also, automated device-to-device communication (e.g. with sensor networks) might become more common in future mobile scenarios. Cross-platform development frameworks need to consider this additional complexity through device management as well as fast and reliable synchronisation which automatically updates other devices based on the current application state.

3.2 Conceptual UI Mapping

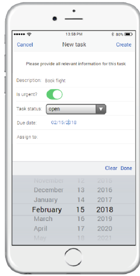
From the current use cases of business apps on smartphones and tablets, some types of user tasks are prevalent. These are mainly related to Create/Read/Update/Delete(CRUD) operations as well as mobile functionalities such as calling a person, receiving notifications, or accessing sensor information (e.g., GPS location). Based on the concepts of *abstract interaction objects* [24] and *presentation units* [5], a mapping is desirable to transform these typical operation components to concrete widget representations of novel smart devices.

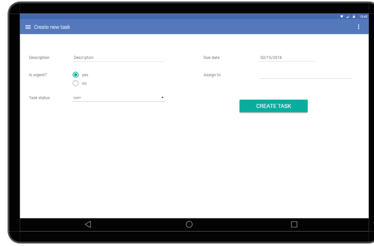
In the following, we describe such a conceptual mapping for business app tasks. It is derived from an analysis of the publicly available design guidelines and best practices by several vendors (see Table 1) which provide app-enabled devices for ubiquitous usage (in contrast to specialised devices such as for cycling). Besides identifying suitable patterns complying to these guidelines, we also keep a generalisable appearance in mind (for inclusion in cross-platform frameworks).

Therefore, possible representations are juxtaposed for traditional mobile devices (smartphones, tablets) and novel app-enabled devices (smartwatches, smart glasses, and smart personal agents). However, to allow for concrete visualisations, we chose representative devices of each class, in particular an iPhone smartphone, an Android-based tablet and smartwatch, and head-mounted Google glass.

Table 1. Analysed design guidelines per device class

Device class	OS guidelines	Representative device
Smartphones/Tablets	Android iOS	Samsung Galaxy S8 iPhone X
Smartwatches	Android Wear/Wear OS watchOS Fitbit Tizen	LG Watch Sport Apple Watch Fitbit Ionic Samsung Gear
Smart glasses	Glass OS HoloLens	Google Glass Microsoft HoloLens
Smart personal assistants	Amazon Alexa Actions on Google	Amazon Echo Google Home





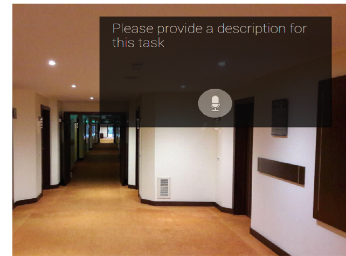


Fig. 1. Possible interface representations for *Create* and *Update* task types

Create and Update. *Create* and *update* interfaces usually share a similar design and are therefore considered together in Fig. 1. On smartphones, create or update steps are usually represented as a list of input fields with corresponding captions and potentially provide contextual help such as placeholder texts or pop-up hints. To better make use of the available space, tablet apps support a multi-column layout, especially when the input fields can be grouped into multiple categories. Due to screen space limitations, smartwatch input can be either represented using a one-column scrolling layout or by a sequence of views per input field. Smart glasses can display the respective fields one by one and are updated using voice controls similar to voice-based smart personal assistants. To allow for a more flexible order of user inputs, advanced techniques might use the concept of frame-based dialogue managers known from chatbot applications

[9]. Using so-called slots for each attribute to be set, the user can provide the required information in any order and the system can focus on asking for missing information to complete the given task.

Select and Read. The *read* operation needs to be distinguished from another related task type: often the user first needs to *select* an item from a collection of objects, before seeing the object's detailed content as depicted in Fig. 2. Therefore, smartphones often provide a scrollable list of items that can be filtered using search keywords and navigated using jump marks. Smartwatches also provide scrollable lists, for example using curved layouts to exploit the round layout of Android-based watches. After selecting an element, individual objects might further be split into different views (e.g., using logical groups of fields) to avoid scrolling behaviour. The same principles apply to smart glasses which also have a limited virtual screen size. Voice-based interfaces may read the list of potential answers or allow users to provide keywords in order to limit the set of results. However, tablets can combine select and read tasks in a single view using the so-called master-detail pattern: The left column of the view presents the list of objects and upon selection updates the right side displaying its content.

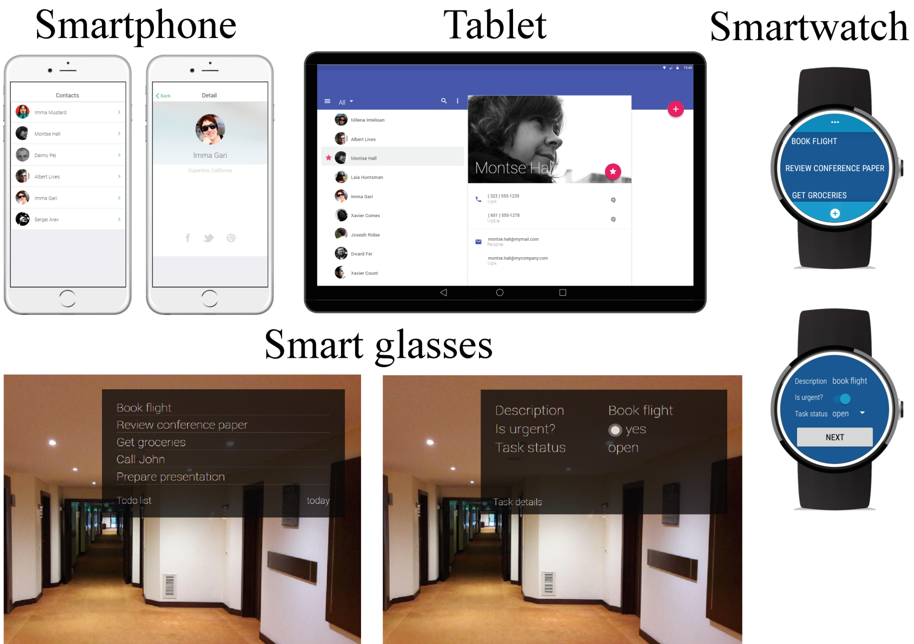


Fig. 2. Possible interface representations for *Select* and *Read* task types

Delete and Modal Pop-Ups. When a *delete* operation is triggered, apps usually require a confirmation to avoid accidental information loss. This confirmation is represented as a modal pop-up view that provides the options to confirm or cancel the current action (in case of deletion) or acknowledge the information prominently displayed on screen. The user must interact with this message in order to continue with his workflow. Therefore, modal views look similar on most screen-based devices and cover the majority of the screen. Voice-based interfaces can instead read out the message and wait for the user to react to it.

Mobile-Specific Tasks. Many mobile-specific tasks have similar representations across screen-based devices. Firstly, *notifications* are present on most devices as unobtrusive information about events when interacting with other apps, either as textual hint at the top or bottom of the screen. In contrast to pop-up messages, notifications disappear automatically or can be closed without blocking the user from performing his actions or waiting for a decision between different options. Voice-based devices can instead read out the message.

Secondly, *phone calls* can be performed not only on smartphones but also on tablets, smartwatches, smart glasses, and smart personal assistants as long as they are equipped with microphone, audio output capabilities, and cellular network connection (potentially indirectly through a connected smartphone). The communication target can usually be chosen by manually dialling a number or selecting someone from a list of contacts (cf. *select* task type) via touch or voice commands.

Finally, *sensor data* may be accessed by specific representations. For example, the GPS location can be visualised by a map on screen-oriented devices. However, this data is commonly contained in attributes of other data objects and therefore already considered in the *read* and *update* task types.

3.3 Modelling Apps Across Device Classes

The model-driven paradigm can provide strong benefits to app development both in terms of development effort regarding the plethora of novel devices as well as the integration and interaction with other applications. Arguably, many concepts from the more established domain of cross-platform development for smartphones can be reused and adapted. To achieve this, the challenges mentioned in Subsect. 3.1 need to be tackled systematically.

Regarding the diversity of input and output characteristics, a high level of abstraction beyond screen-oriented UIs is mandated, for example using declarative notations for representing use tasks. Consequently, arbitrary platforms can be supported by developing respective generators which implement a suitable mapping from descriptive models to platform-specific implementations.

Ideally, a "one model fits all platforms" approach can therefore be achieved by applying two types of transformations that do not modify the content but adapt the task appearance to different supported devices:

1. On the one hand, 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, if necessary, leave out complementary details. In the inventory management example, the round design of a smartwatch can be exploited by the curved list layout such that content readability is improved and screen usage is maximised.
2. On the other hand, the content structure can be *re-formatted* by an adaptive UI according to usual platform interaction patterns, e.g., let the user scroll through large amounts of information, present it in multiple subsequent steps, or as a hierarchical structure providing more details on request [4,5].

A high degree of abstraction can be achieved for user interactions by modelling user inputs in terms of *intended actions* for completing a particular task. An intermediate mapping layer can then transform actual inputs to the respective actions. For instance, a "back" action can be linked to a hardware button, displayed in a navigation bar on screen, bound to the right-swipe gesture (as recommended by the Wear OS guidelines [10]), or recognized by a spoken keyword in smart personal assistants. Another example is the usage of *default actions* to navigate through the app. Whereas in iOS, one possible action can be displayed in the top-right corner of the navigation bar, Wear OS uses so-called *Action-Drawers*[3] which propose one or more possible actions from the bottom edge of the screen.

Business apps are usually designed to support specific goal-oriented workflows which can be decomposed into individual tasks. This perspective of a user task model aligns with the desired high level of abstraction [22]. Subsequently, two task-oriented notations are presented that embody the task-oriented approach. The *ConcurTaskTree (CTT)* notation consists of three main elements [16]:

– the abstract *task* descriptions which together form the use case's functionality,
– temporal *operator*s defining the allowed sequences of executed tasks, representing an abstract notion of navigation actions within a use case, and
– the user *role*s (per task) that are allowed to interact with the system

CTTs have been refined to suit user interface development through decomposition over multiple levels [17]. An exemplary model suited for interface generation is presented in Fig. 3. The example shows an excerpt of a simple inventory management task in which the user (either a warehouse clerk or product manager) first selects an item from a list (with the possibility to filter the list content by title) and gets presented the item details (with data on title, description, pictures, price, product category, and quantity on stock; collapsed in Fig. 3). Depending on the user's role, different modes of editing the inventory are possible before concluding or aborting the process. Warehouse clerks can enter the quantity and position of newly arrived item replenishments. Product managers may instead update the item master data.

[3] https://designguidelines.withgoogle.com/wearos/components/action-drawer.html.

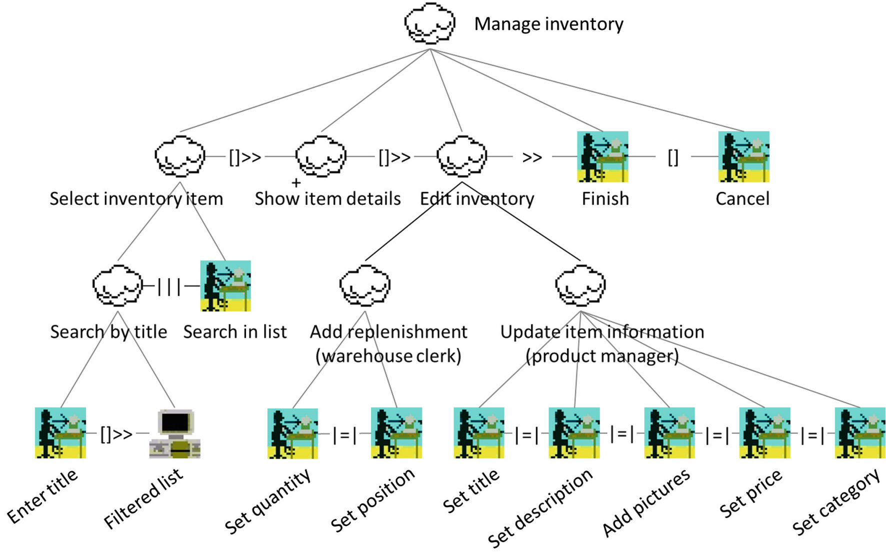


Fig. 3. Sample CTT model for an inventory management use case

As can be seen, the detailed decomposition at operational level creates a tree with user tasks on multiple levels of abstraction which diminishes the overview about the high-level tasks and particular user interactions. In addition, still many aspects such as navigation, item groups, informative labels, or object structures are not explicitly contained in the notation and need to be provided in separate models (e.g., using UML class diagrams for data models) to enable a fully automated generation of user interfaces.

Previously, we have proposed the graphical Müunster App Modeling Language (MAML) for specifying business apps based on five main design goals [18]:

- *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 modelling* specifies the application domain but also sets a high level of abstraction by interpreting data manipulation as a process.
- *Modularisation* of activities in distinct use cases helps for maintenance, especially for domain experts.
- *Declarative description* of the complete app, including necessary specifications of data model, business logic, user interactions, and UI views.

Compared to CTTs, the same exemplary scenario is depicted in Fig. 4 as MAML model. In MAML, this task is modelled in a *use case* as depicted in Fig. 4. The model contains a sequence of activities, from a *start event* (labelled with (1) in Fig. 4) towards one or several *end events* (2). In the beginning, a *data source* (3) specifies the data type of the manipulated objects and whether

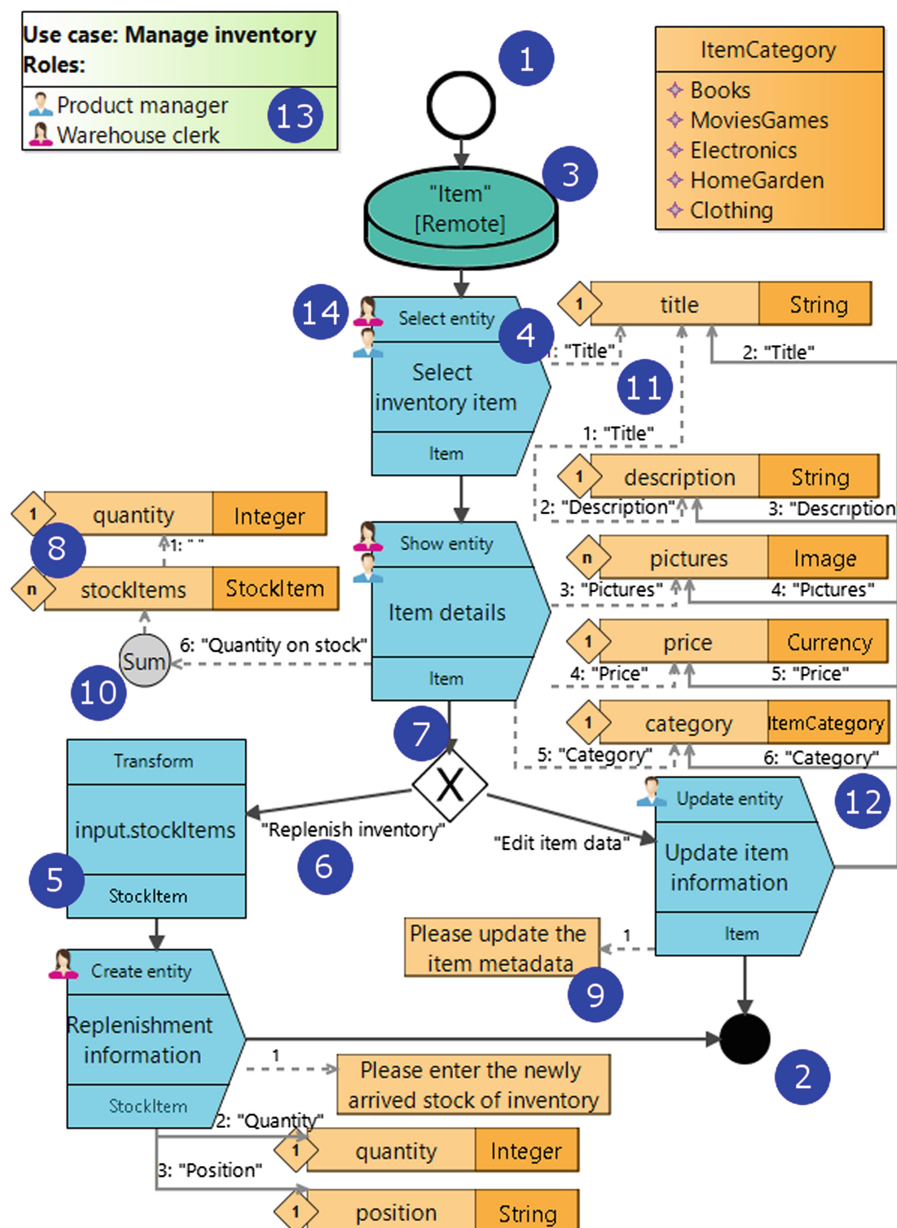


Fig. 4. Sample MAML model for an inventory management use case

they are only saved locally on the device or managed by the remote backend system. Data can then be modified through a pre-defined set of (arrow-shaped) *interaction process elements* (4), for instance to *select/create/update/display/delete entities*, show *pop-up messages*, or access device functionalities such as the *camera* and starting a phone *call*. This describes the desired user actions on a high level and can be mapped to device-specific UI representations as described in Subsect. 3.2. *Automated process elements* (5) represent invisible processing steps without user interaction, for example *including* other models for process reuse, or navigating through the object graph (*transform*). The navigation between connected process steps happens along the *process connectors* (6) which can be supplied with captions. To account for non-linear workflows, process flows can be branched out using *XOR* elements (7).

In contrast to CTTs, MAML models additionally contain the data objects which are displayed within a respective process step. *Attributes* (8) consist of a cardinality indicator, a name, and the respective data type. Besides pre-defined data types, custom types can be defined and further described through nested attributes. *Labels* (9) provide explanatory text, and *computed attributes* (10) are used to calculate derived values at runtime based on other attributes. In MAML, only those attributes are modelled which will be used in a particular process step, for instance for including it in the graphical user interface.

Two types of connectors exist for attaching data to process steps: Dotted arrows represent a *reading relationship* (11) whereas solid arrows signify a *modifying relationship* (12) with regard to the target element. Every connector which is connected to an interaction process element also specifies an order of appearance and a field description. For convenience, multiple connectors may point to the same UI element from different sources (given their data types match). Alternatively, to avoid confusing connections across larger models, UI elements may instead be duplicated to different positions in the model and will automatically be matched at runtime.

Finally, MAML supports a multi-role concept to define (13) and annotate (14) arbitrary role names to the respective interaction process elements because many business processes such as approval workflows involve different people or departments.

It can be observed that the domain-specific MAML notation allows for a more concise and clearly arranged representation of the app content, for example by offering different task types which encapsulate the respective semantics as opposed to the more general CTT models. At the same time, MAML models contain precise technical details on the resulting app, such that fully functional apps can be generated from the notation. Using the presented mapping approach, apps can be flexibly generated from the same MAML models both for smartphones (Android and iOS) as well as Wear OS smartwatches.

4 Discussion

To put the presented approach in a broader context, we discuss two scenarios that underline the aforementioned issues and potential solution.

The first scenario in the *logistics* domain with an inventory management app was already depicted in Figs. 3 and 4. It represents a typical business process that can be performed by a single user when interacting with a centralised back-end system. However, it proves the potential for MDSD in generating apps for very different devices simultaneously from a common model: Firstly, a warehouse clerk might want to use smart glasses in order to have hands-free interaction with the inventory system when replenishing items. A product manager might instead use a smartphone to update erroneous information while inspecting shelves. Achieving business app development using model-driven techniques therefore seems particularly beneficial with regard to the flexibility it provides towards the future end user. This freedom to choose an appropriate device given the current usage context or personal preferences can not only propel technology acceptance but also allows for more experimentation with different modes of work.

As a second scenario, consider a *health app* which is integrated with a patient information system. Before making an appointment, patients can be asked to give more details about the health issue they are experiencing, for example through standardised questionnaires. This information is passed to the doctor to support his diagnosis. In order to follow-up on the medical treatment, sensors can regularly monitor vital parameters over a specified time interval and report the results back to the medical office such that doctor's assistants can take action when observing abnormal values.

This complex workflow involves multiple actors who need to share information over a longer period of time. In addition, different devices are combined according to their capabilities (e.g., wearable devices for sensing heart rate). However, tasks related to data manipulation might be limited on some devices, e.g., entering lots of data on a smartwatch is tedious. Using automated transformations of user interfaces can alleviate the problem only to a certain degree. Most probably, a co-existence of apps for multiple devices in parallel will be more useful than forcing all tasks to be performed on the same device. Again, the model-driven paradigm can provide significant benefits when offering a broad range of devices without linear increase of development efforts. However, to achieve the sketched multi-device usage, the potentially concurrent interaction between multiple devices requires a reliable and fast synchronisation of data among the respective devices. Different techniques such as Operational Transformation (OT) and Conflict-Free Replicated Data Types (CRDT) have been proposed for mobile data synchronisation but further research is necessary to apply them to the context of business apps [7,20]. In both scenarios, the diversity of user interfaces is best tackled by transforming a high-level and platform-agnostic model that relies on abstract *presentation units* which are later mapped to *concrete interaction objects*, i.e. suitable widget representations for platform-specific look-and-feel [5,24].

When putting the presented mapping into practice, the CTT representation as a general purpose notation for user tasks is not sufficient. Additional notations are required to specify the data model and possible interaction and

navigation flows within an app. Other cross-platform approaches targeted to smartphones such as BiznessApps [2] and Bubble [3] provide graphical configurators for simplified app modelling, however, their screen-oriented approach falls short of covering wearable UIs. Similarly, user interaction standards such as the Interaction Flow Modeling Language (IFML) are implicitly tied to screen-based output [15]. In contrast, domain-specific languages such as MAML can provide components on a high level of abstraction while at the same time integrating UI and interaction aspects. Apart from being useful for holistically modelling apps, the MAML framework also includes code generators that output fully functioning smartphone apps. To demonstrate the feasibility of deriving a suitable representation from the same input models, the framework was extended by implementing a generator for Android Wear 2.0 smartwatches (Figs. 1 and 2 contain actual app screenshots).

The proposed high-level modelling approach for wearable app specifications also caters for the limitations of this paper regarding the future evolution of the field. Whereas differences in smartphone UIs have assimilated over time to a large degree (at least with regard the main input and output components), the chosen representations in Subsect. 3.2 can only represent a small subset of the various UIs and interaction patterns which already exist or maybe emerge in the future for wearable devices. The novelty of this highly dynamic field of wearables will certainly entail several changes to typical platform characteristics regarding both hardware and software. Vendors continuously explore interfaces and interaction possibilities, heavily modifying their best practice guidelines when presenting new versions of their platform. At the same time, this paper only scratches the surface of cross-platform development complexities across multiple device classes. In-depth research needs to address each device class individually which still exhibits a high variability of devices capabilities – future consolidation across vendors is possible but not foreseeable anytime soon.

5 Conclusion and Outlook

In this paper, we have investigated the challenges and potential solutions for applying model-driven techniques to the development of business apps for novel app-enabled devices. In particular, user interface related challenges currently limit the extension of established cross-platform techniques to these new classes of heterogeneous devices. We exemplify how the typical building blocks of business apps might be generically mapped to smartphones, tablets, smartwatches, and smart glasses.

In order to incorporate these in a model-driven approach, task-oriented modelling – without actually specifying the concrete user interface – allows for a suitably high level of abstraction. We deem this approach promising to bridge the heterogeneity of devices and enable fast development of apps that are flexibly targeted to a broader range of devices through adaptive layouts as well as device class specific reformatting of contents. Although both the MAML notation and CTT utilise a task-oriented approach, we argue that domain-specific languages

such as MAML might be more suitable to modelling user interfaces than general purpose modelling notations due to the alignment with the target domain, thus condensing domain concepts more clearly to a high level of abstraction.

However, the current situation leaves room for further research in different directions such as observing emerging commonalities in user interfaces of the individual device classes that are subsumed by the umbrella term "wearables", and technical hurdles to synchronise content and application state in the dynamic interplay of mobile devices. Furthermore, the implementation of the presented concepts in an actual model-driven framework constitutes ongoing work of the authors by extending the MAML cross-platform framework with support for further novel device classes using the presented mapping approach.

References

1. Alulema, D., Iribarne, L., Criado, J.: A DSL for the development of heterogeneous applications. In: FiCloudW, pp. 251–257 (2017)
2. Bizness Apps: Mobile app maker – bizness apps (2018). http://biznessapps.com/
3. Bubble Group: Bubble - visual programming (2018). https://bubble.is/
4. Eisenstein, J., Vanderdonckt, J., Puerta, A.: Applying model-based techniques to the development of UIs for mobile computers. In: IUI (2001)
5. Eisenstein, J., Vanderdonckt, J., Puerta, A.: Applying model-based techniques to the development of UIs for mobile computers. In: Proceedings of the 6th International Conference on Intelligent User Interfaces, pp. 69–76. ACM (2001)
6. El-Kassas, W.S., Abdullah, B.A., Yousef, A.H., Wahba, A.: ICPMD: integrated cross-platform mobile development solution. In: ICCES (2014)
7. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: SIGMOD, pp. 399–407. ACM (1989)
8. Esakia, A., Niu, S., McCrickard, D.S.: Augmenting undergraduate computer science education with programmable smartwatches. In: Decker, A., Eiselt, K., Alphonce, C., Tims, J. (eds.) SIGCSE, pp. 66–71. ACM (2015)
9. Goddeau, D., Meng, H., Polifroni, J., Seneff, S., Busayapongchai, S.: A form-based dialogue manager for spoken language applications. In: ICSLP, pp. 701–704 (1996)
10. Google LLC: Google developers. https://developers.google.com/
11. Jesdabodi, C., Maalej, W.: Understanding usage states on mobile devices. In: International Joint Conference on Pervasive and Ubiquitous Computing, pp. 1221–1225. ACM (2015)
12. Koren, I., Klamma, R.: The direwolf inside you: end user development for heterogeneous web of things appliances. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) ICWE 2016. LNCS, vol. 9671, pp. 484–491. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38791-8_35
13. Majchrzak, T.A., Ernsting, J., Kuchen, H.: Achieving business practicability of model-driven cross-platform apps. OJIS **2**(2), 3–14 (2015)
14. Neate, T., Jones, M., Evans, M.: Cross-device media: a review of second screening and multi-device television. Pers. Ubiquit. Comput. **21**(2), 391–405 (2017)
15. Object Management Group: Interaction flow modeling language (2015). http://www.omg.org/spec/IFML/1.0
16. Paternò, F.: Model-Based Design and Evaluation of Interactive Applications. Springer, London (2000). https://doi.org/10.1007/978-1-4471-0445-2

17. Pribeanu, C.: An approach to task modeling for user interface design. In: Proceedings of the 3rd World Enformatika Conference, vol. 5, pp. 5–8 (2005)
18. Rieger, C., Kuchen, H.: A process-oriented modeling approach for graphical development of mobile business apps. Comput. Lang. Syst. Struct. **53**, 43–58 (2018)
19. Rieger, C., Majchrzak, T.A.: Conquering the mobile device jungle: towards a taxonomy for app-enabled devices. In: WEBIST, pp. 332–339 (2017)
20. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24550-3_29
21. Singh, K., Buford, J.: Developing WebRTC-based team apps with a cross-platform mobile framework. In: Consumer Communications and Networking Conference (2016)
22. Sinnig, D., Chalin, P., Khendek, F.: Common semantics for use cases and task models. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 579–598. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_30
23. van der Meulen, R., Forni, A.: Gartner says worldwide wearable device sales to grow 17 percent in 2017 (2017). https://www.gartner.com/newsroom/id/3790965
24. Vanderdonckt, J.M., Bodart, F.: Encapsulating knowledge for intelligent automatic interaction objects selection. In: Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems, pp. 424–429. ACM (1993)