Synthesis of mobile applications using AgileUML

K. Lano
L. Alwakeel
kevin.lano@kcl.ac.uk
lyan.alwakeel@kcl.ac.uk
Dept. of Informatics, King's College London
London, UK

S. Kolahdouz-Rahmimi
Dept. of Software Engineering
Isfahan, Iran
sh.rahimi@eng.ui.ac.ir

H. Haughton
Holistic Risk Solutions Ltd
London, UK
howard.haughton@gmail.com

ABSTRACT

In this paper we describe a method to apply the AgileUML toolset to synthesise mobile apps from UML models. This is a lightweight model-driven engineering (MDE) approach suitable for app developers who need to rapidly produce native apps for either or both Android or iOS platforms.

In contrast to other MDE approaches for app development, the approach aims to minimise the extent of manual effort by using very concise high-level specifications which abstract from technical details as far as possible, while still providing explicit definitions of functional app behaviour.

CCS CONCEPTS

• **Software and its engineering → Application specific development environments**.

KEYWORDS

MDE; Mobile apps; Agile development

ACM Reference Format:
K. Lano, L. Alwakeel, S. Kolahdouz-Rahmimi, and H. Haughton. 2021. Synthesis of mobile applications using AgileUML. In *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference) (ISEC 2021), February 25–27, 2021, Bhubaneswar, Odisha, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3452383.3452409

1 INTRODUCTION

Mobile apps have become a key means by which users access a wide range of software services such as online shopping, banking, mapping and location services and social media. There are over 2 million apps on each of the main app stores (Google Play for Android and Apple iTunes for iOS).

However, development for Android or iOS can be a complex and time-consuming activity, due to the specialised programming mechanisms and libraries of each platform, and the impact of evolution and diversification within these platforms. For example, in the case of iOS, apps can be developed based on either the Swift or Objective-C languages, and using either the UIKit event-based approach, or the SwiftUI data-centered approach. In the case of Android either Kotlin or Java programming languages can be used, and differing levels of support provided for devices running older versions of the OS. Despite the facilities offered by the respective app development toolsets (Android Studio and Xcode), manual development for either platform requires a high level of specialised expertise, comparable to that needed for enterprise information system (EIS) platforms such as Java EE or .Net.

There are some similarities between mobile app technologies and EIS technologies (for example, systems in both domains should separate UI aspects from functional aspects, and need to manage persistent data and interaction with remote services), but there are also specific issues in mobile apps. Both network connections and data rates may be variable and intermittent, leading to requirements to cache data to ensure some functional capabilities in offline situations. Limited device processing and memory resources also encourage the use of caching to avoid unnecessary network access, and the use of asynchronous or off-device processing for computationally-intensive tasks. Restricted screen size on mobile devices leads to the need to simplify interfaces and user interaction.

Model-driven engineering (MDE) has been applied to the problem of accelerating app development, for example in the MD^2 and PIMAR methods [12, 31]. MD^2 also takes a lightweight MDE approach to app specification, via the use of simple textual models of app UI, data and behaviour, but it is focussed on data-driven business apps for tablets. PIMAR uses graphical process models and class diagrams, and requires significant MDE expertise. Both of these approaches use the familiar MVC architecture for app front ends, but do not incorporate other app patterns. They do not appear to support the specification of detailed customised functionality at the modelling level – instead such functionality must be written manually in the generated app code.

In our approach, we adopt the existing lightweight MDE techniques of the AgileUML toolset [8]. AgileUML uses a simplified UML subset (UML-RSDS [16]) to define the data and behaviour of applications via class diagrams and use cases with behaviour defined by pseudocode statements or OCL postconditions. A purely textual specification of classes and use cases can also be written, in KM3 notation [14]. For mobile apps, we provide a predefined set of components representing services such as authentication (*FirebaseAuthenticator*), data retrieval from remote services (*InternetAccessor*) and graph display (*GraphDisplay*). These components and their operations can be utilised within app specifications.

ISEC 2021, February 25–27, 2021, Bhubaneswar, Odisha, India
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9046-0/21/02…$15.00
https://doi.org/10.1145/3452383.3452409

From app specifications, complete functional code can be generated in a variety of implementation languages (Java, C#, C++, C, Python, etc). An emphasis is placed on code efficiency and correctness-by-construction. AgileUML already synthesises Java web systems using JSPs and Servlets [17]. These use the MVC UI pattern, and EIS patterns such as Session Facade, Data Access Object (DAO) and Value Object (VO) [9]. We also adopt these patterns for Android and iOS app architectures, together with app-specific patterns such as cached DAOs, and Service Activator.

We also take account of more advanced app patterns such as VIPER and MVVM, and build in UI and security best practices. For example, we avoid the use of forward/backward navigation (akin to following links in a web interface), instead lateral navigation using tabs is enforced. We also prefer tabs to the use of *hamburger menus*, which conceal the available command options [24]. To improve security and data integrity, all data entered by a user has to pass through a validation bean before being submitted to the back end of the app. Standard security checks, such as the absence of SQL injection code in string data, can be defined by OCL conditions. Features can be stereotyped as ≪*password*≫ or as ≪*hidden*≫ so that they are obscured on data entry, and are not displayed on list screens. Login functionality can be specified using cloud-based authentication via Firebase[1]. By using the *FirebaseAuthenticator* component, access to particular functionalities can be restricted to logged-in users. Use case authorisation levels could be introduced, so that only users with sufficient authorisation can utilise functionalities marked as restricted.

In general, the advantages of MDE for mobile apps are:

- A substantial reduction in the manual effort required for defining layouts, and the code for data management and UI interaction.
- Systematic architectures can be imposed on apps, following best practices for app architectural patterns.
- The same functionality can be implemented for multiple platforms and versions (eg., for Android, iOS/UIKit and iOS/SwiftUI) without additional effort.
- Apps can be quickly modified and extended by changing their specification and re-generating their code.
- App test cases can be generated from the app specifications (model-based testing).

We target general-purpose apps, but exclude game apps, which require specialised modelling approaches [7]. Our specific focus is in financial and health apps, which typically involve complex custom functionality with high efficiency and security requirements [1, 13].

The specific goals of the AgileUML approach for mobile apps are:

(1) Minimise the extent of manual coding needed;
(2) Simplify user interaction and minimise the number of screens needed;
(3) Optimise efficiency via the use of caching and asynchronous execution of time-consuming tasks where possible;
(4) Optimise modularity via separation of app code into tiers (client, presentation, business, integration and resource tiers) and cohesive components.

[1]https://firebase.google.com

Section 2 summarises AgileUML and UML-RSDS. In Sections 3, 4 and 5 we describe in detail the app synthesis approach used, Section 6 gives an evaluation on case studies, Section 7 compares our approach to related work, and Section 8 discusses limitations and future work.

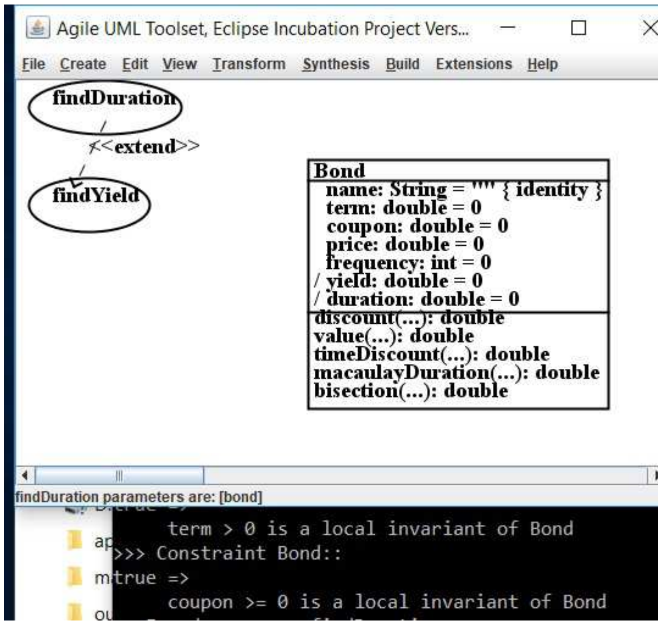
2 AGILEUML AND UML-RSDS

UML-RSDS is a subset of UML 2.5, consisting of class diagram notation, use cases, a subset of OCL 2.4, and pseudocode text notation for activities. It is more declaratively oriented than other executable UML approaches such as fUML [23], and permits complete applications to be specified using high-level representations (class diagrams, OCL and use cases), close to requirements. OCL constraints are automatically mapped to pseudocode activities via a "generate design" process. Alternatively, activities may be directly written in the specification.

In textual KM3-based notation, a class definition can include definitions of stereotypes, invariants, data features and operations of the class:

```
class C extends D
{ stereotypes;
  invariants;
  attributes and references;
  operations
}
```

Stereotypes are either single strings such as "interface" or "abstract" to mark the category of the class, or are tag-value pairs such as "author='kcl'". Stereotypes are also written in the form ≪*stereotype*≫, eg., ≪*persistent*≫ to mark a class as persistently stored.



```
findDuration parameters are: [bond]
      term > 0 is a local invariant of Bond
>>> Constraint Bond::
true =>
      coupon >= 0 is a local invariant of Bond
```

Figure 1: Bond app class diagram

For example, the text version of the *Bond* class in Figure 1 has the form:

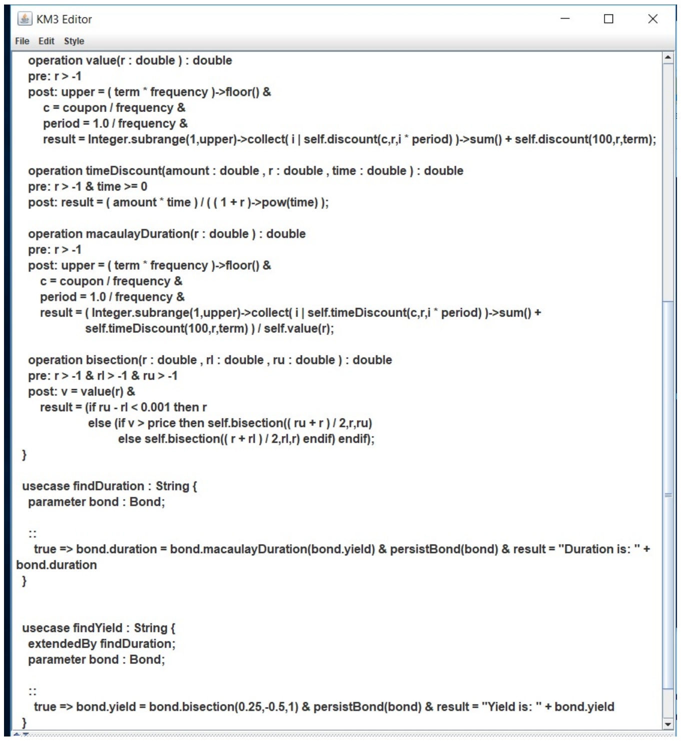
```
class Bond
{ invariant term > 0;
  invariant coupon >= 0;

  attribute name identity : String;
  attribute term : double;
  ...

  operation discount(amount: double, rate: double,
    time: double) : double
  pre: r > -1
  post: result = amount/((1+r)->pow(time));

  ...
}
```

This class represents fixed-interest rate financial products such as UK government gilts. Class invariants can be used to construct and check test cases for the class, and as the basis of validator components for use cases that modify instances of the class. Operations are specified with pre and post conditions in OCL format. The preconditions can also be used to guide test case production, and are checked at execution time. An alternative approach is to specify an operation *activity* in pseudocode. Further examples of operation specifications are given in Figure 2. Note that *bisection* is recursively defined.



```
  operation value(r : double ) : double
  pre: r > -1
  post: upper = ( term * frequency )->floor() &
     c = coupon / frequency &
     period = 1.0 / frequency &
     result = Integer.subrange(1,upper)->collect( i | self.discount(c,r,i * period) )->sum() + self.discount(100,r,term);

  operation timeDiscount(amount : double , r : double , time : double ) : double
  pre: r > -1 & time >= 0
  post: result = ( amount * time ) / ( ( 1 + r )->pow(time) );

  operation macaulayDuration(r : double ) : double
  pre: r > -1
  post: upper = ( term * frequency )->floor() &
    c = coupon / frequency &
    period = 1.0 / frequency &
    result = ( Integer.subrange(1,upper)->collect( i | self.timeDiscount(c,r,i * period) )->sum() +
               self.timeDiscount(100,r,term) ) / self.value(r);

  operation bisection(r : double , rl : double , ru : double ) : double
  pre: r > -1 & rl > -1 & ru > -1
  post: v = value(r) &
    result = (if ru - rl < 0.001 then r
                else (if v > price then self.bisection(( ru + r ) / 2,r,ru)
                        else self.bisection(( r + rl ) / 2,rl,r) endif) endif);
 }

 usecase findDuration : String {
  parameter bond : Bond;

  ::
   true => bond.duration = bond.macaulayDuration(bond.yield) & persistBond(bond) & result = "Duration is: " +
bond.duration
 }


 usecase findYield : String {
  extendedBy findDuration;
  parameter bond : Bond;

  ::
   true => bond.yield = bond.bisection(0.25,-0.5,1) & persistBond(bond) & result = "Yield is: " + bond.yield
 }
```

Figure 2: Bond app operations and use cases

Use cases also have a textual format with the general form:

```
usecase Uc : resultType
{ extend/include relationships;
  parameter declarations;
  stereotypes;
  invariants;
  local attributes;
  local operations;
  preconditions;
  postconditions;
}
```

UML use cases combine aspects of classes and operations, they can also be related by extension and inclusion dependencies. Stereotypes can indicate the actor of the use case, or that a use case is private, etc. Use cases represent global operations of the system, and their behaviour is defined by a series of postconditions which can create/delete instances of classes, invoke operations on instances, invoke included use cases or local operations, and display information to the user. The definitions of the use cases $findYield$ and $findDuration$ are given in Figure 2.

The AgileUML toolset [8] supports the construction, analysis and implementation of UML-RSDS specifications, via a set of code generators (for Java, C#, C++, C, Python, etc). An important feature of AgileUML is the ability to add new code generators using the $\mathcal{CSTL}$ specification notation [19]. This enables code generators to be defined by concrete syntax text-to-text rules, so that users can add a new code generator without knowing the abstract syntax (metamodels) of the target language or of UML-RSDS. Code generators for Swift 5 and for Java 8 are defined in this manner, and are used to generate iOS and Android apps.

Use cases represent the application-level functionalities or services offered by a system. There are several kinds of UML-RSDS use case which can be defined:

- General use cases, which can either be ≪*private*≫ or ≪*public*≫ (the default).
- EIS use cases, representing CRUD operations on class data. These are public.

General use cases can depend on each other via *include* and *extend* relationships. An included use case represents a subtask within the including use case, whilst an extension use case represents a variation or additional functionality dependent upon the main use case. Eg., in Figure 1, $findDuration$ is an extension of the main $findYield$ use case. One use case can be directly or indirectly extended by any number of extension cases. We refer to the main use case of such a cluster of extensions as the *primary* use case.

Use cases are implemented in a *ModelFacade* component, which contains their generated code. The *ModelFacade* is the central point of access to business tier functionality and data.

Table 1 summarises the use of UML-RSDS elements to express mobile app elements.

3 APP SYNTHESIS

Android and iOS are completely predominant in the mobile OS market, with Android having over 80% of the market and iOS around 15% of the market[2]. Thus we provide support only for these platforms.

[2]Gartner Inc, press release 23rd May 2017.

UML-RSDS Element	*Mobile app Element*
≪*public*≫ use case with postconditions or activity	UI screen with ViewController, validation bean, and *ModelFacade* operation
≪*private*≫ use case with postconditions or activity	*ModelFacade* operation representing callback or internal UI event
Use case *extend* relationship	Grouping of related use cases on one screen
Use case preconditions or class invariants	Validation checks in validation bean
≪*persistent*≫-stereotyped classes	SQLite database table and DAO
≪*cloud*≫-stereotyped classes	Cloud datastore and DAO
≪*remote*≫ classes	*InternetAccessor* and cached DAO
Predefined components	Utilities and platform services
Customised components	Non-platform/specialised services

Table 1: Conceptualisation of mobile app elements

The *Build* menu of the AgileUML Version 2.0 toolset [8] provides three options for generating mobile apps from UML-RSDS specifications:

(1) For Android (Versions 7+, SDK API level 25+), using Java 8.
(2) For iOS (versions 10+), using UIKit interfaces with Swift 4 or 5.
(3) For iOS (versions 12+) using SwiftUI.

In each case, a common five-tier architecture is used, with UI screens in a client tier, view controllers[3] and validation beans in the presentation tier, and *ModelFacade* (session facade) and entity bean classes to implement business tier functionality. Further data access object and service activator classes are also defined for the integration tier. Value object classes are defined to support transfer of data between tiers. The resource tier can include files, databases and remote resources accessed via URLs (Figures 3 and 4 show the schematic architectures for Android apps).

To initiate app synthesis, a specification class diagram and use case model is created or loaded into AgileUML. Figures 1, 2 show an example of a loaded specification. Selecting the *Android* or *iOS* options on the *Build* menu will then generate code and layout artifacts of the app in appropriate subdirectories, mimicking the structure of an Android Studio or Xcode app. A manifest file defining appropriate app permissions is also generated for Android, together with a gradle app build file. For iOS, a podfile is generated in cases where external libraries need to be incorporated into the build.

3.1 Data and resources

There is a key distinction between persistent and non-persistent classes. If E is a non-persistent class, then conventional Java or Swift code is generated for E, and use cases can use its instances

[3]For SwiftUI, view controllers are omitted.

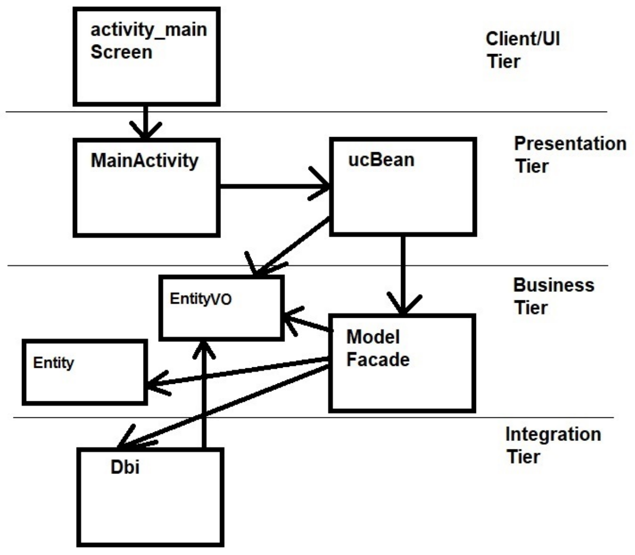


Figure 3: Architecture for single-screen Android apps

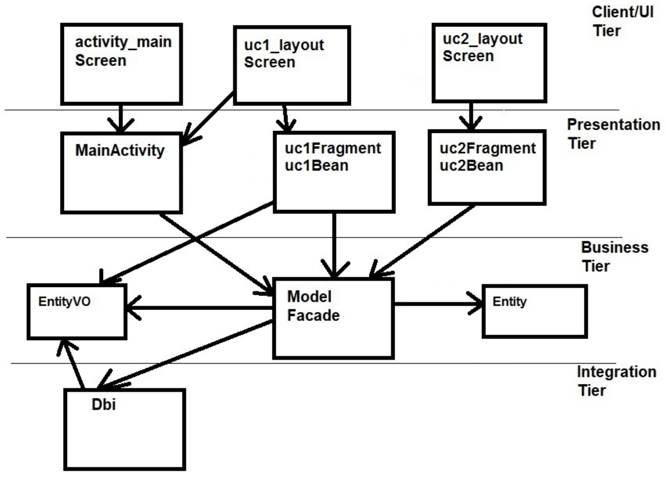


Figure 4: Architecture for multiple-screen Android apps

and operations, but these instances are not persisted. Normally, E should have a String-typed primary key, so that its instances can be referenced in the specification by the notation $E[id]$, denoting the instance of E with primary key value id. In the case of persistent classes, an SQLite database is used to store their instances, and the class must have a primary key. Data is transferred into and out of the database by a DAO component, $Dbi.java$ or $Dbi.swift$, invoked by the *ModelFacade*. For each persistent class E, the DAO component supports operations $createE$, $deleteE$, $editE$, $listE$, $searchByEatt$ to create, delete, modify and list instances of E. Synchronisation of E instances in runtime memory with the database is not performed automatically, instead the operation $loadE()$ can be called to load all database instances of E into runtime memory, and $persistE(ex)$ can be called to update the database data of a specific instance $ex : E$ – the database must already hold a row for the key of ex.

Remote data sources can be represented in the specification as classes E with the stereotype ≪*remote*≫. A cached DAO is used to manage ≪*remote*≫ data sources. For each ≪*remote*≫ E, the component E_DAO will be automatically included in the specification, and its operations can be invoked (eg., as $E\_DAO.parseCSV(data)$) from use cases. Operations are provided for decoding CSV[4], XML and JSON data into E instances, checking the cache, and deriving a URL for specific data retrievals. A base URL string for the remote data source should be specified as a stereotype url = "*baseurl*" of E. An $InternetAccessor$ predefined component is also included, which provides asynchronous invocation of Internet connections using $HttpURLConnection$ on Android or $URLSession$ on iOS. The $ModelFacade$ is a listener/delegate for callbacks *internetAccess Completed* issued by the component. Thus typically the specifier will need to define the actions to be taken when the connection session is completed, by writing a specification for the private $internetAccessCompleted$ use case.

Figure 5 shows the schematic architecture used for remote data sources, for Android apps.

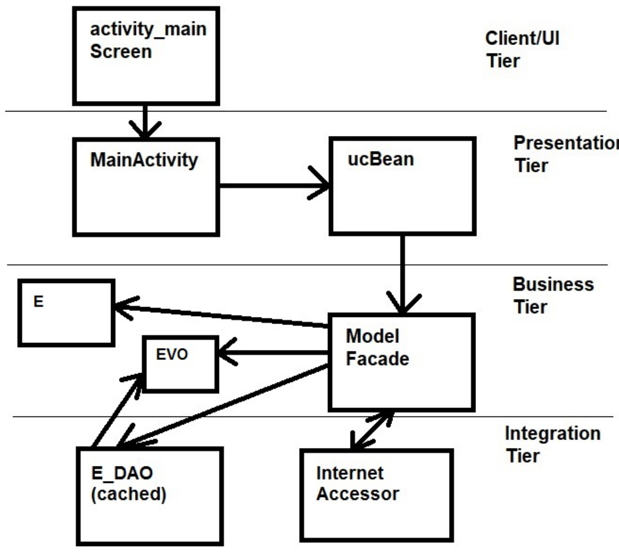


Figure 5: Architecture for remote data sources

An alternative to locally-persisted data is to define an entity with the ≪*cloud*≫ stereotype. This is then persisted via the Firebase cloud realtime database. Local entity data is synchronised with the cloud data via the Firebase update notification mechanism.

[4]In finance, datasets are often available in CSV format, eg., Yahoo Finance exchange rate and share price daily data.

3.2 Predefined and custom components

Remote or local services used by the app are represented in the specification as ≪*component*≫ classes, possibly accompanied by associated callback interfaces. Utilities (such as $DateComponent$) and wrappers for platform services (such as $InternetAccessor$, $FileAccessor$) are defined as predefined components: their specifications cannot be changed, and code is automatically generated for them for each platform utilising the built-in APIs. Thus these components are adapters, providing a uniform specification of operations for common services which are available in different forms on different platforms. They are usually singleton or static classes.

A singleton component C is used by use case uc via the activity code:

```
var cinst : C ;
cinst := C.getInstance() ;
cinst.op()
```

If C has a $CCallback$ interface, then uc can set its host class (the model facade) as the delegate for callbacks:

```
cinst.setDelegate(self)
```

The private use cases representing the callback operations would also be specified. Whenever C is added to the system explicitly or implicitly, $CCallback$ is added as an interface of the $ModelFacade$, and the callback operations are added as private use cases of the system. Components $ImageDisplay$, $GraphDisplay$ and $WebDisplay$ are used only as result types of use cases, to display results as images, graphs or web pages.

Other components are defined as customised components, with user-defined specifications and code. This enables flexible customisation of service adapters. For example, $MediaComponent$, $PhoneComponent$ and $SMSComponent$ have KM3 specifications and supplied Android and iOS implementations.

4 ANDROID APPS

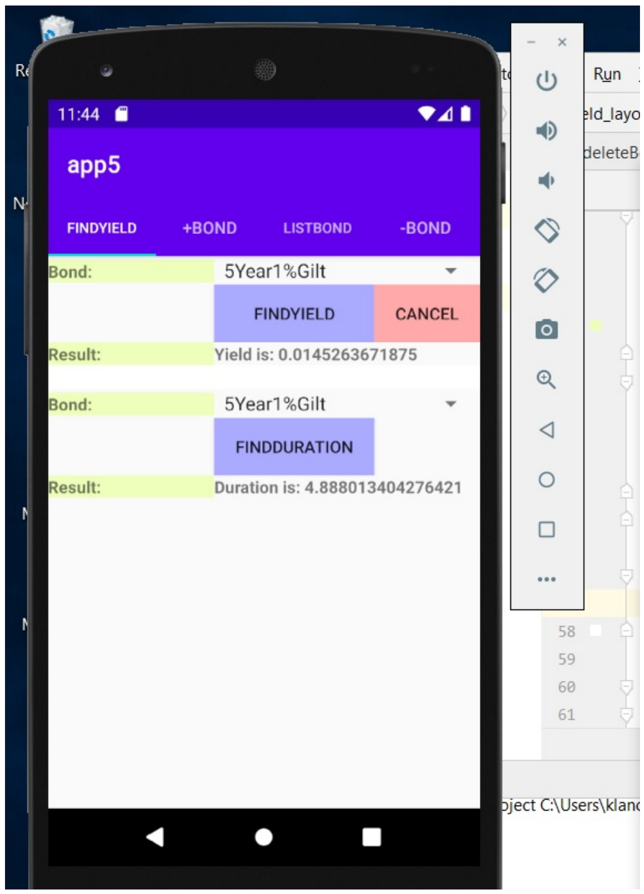
From system specifications, both single-screen and multi-screen Android apps can be generated:

- Single screen apps are specified using a single public primary use case together with possibly one or more extension use cases. The parameters of the primary use case become input fields of the screen, and there is also a field for its result. Extension use cases can access these input parameters and the results of the primary use case via attributes of the main use case. A stereotype $image$ = "$file$" of the main use case defines an image for the app screen.
- Multi-screen apps are specified by two or more independent public use cases (not related by *extend* or *include* dependencies). Each use case becomes a screen within a tabbed array of screens. Images can be nominated for the independent use cases.

All the specification use cases become operations in the business tier of the app, however only public general use cases and EIS use cases have visible screens in the UI of the app.

Private use cases are typically either (i) used to represent common functionality called from several *including* use cases, or (ii) represent callbacks from asynchronous services/resources, or operations called from interactive UI components such as maps.

Public use cases should not normally be included in other use cases, however they can be extensions of other public use cases. A group of a primary use case and its extensions are treated as closely-related functionalities to be represented together on a single app screen. Eg., Figure 6.



Bond: 5Year1%Gilt

Result: Yield is: 0.0145263671875

Bond: 5Year1%Gilt

Result: Duration is: 4.888013404276421

Figure 6: Multi-screen Android app

4.1 Single-screen apps

In the single-screen case, a public general use case *uc* with input parameters $p_1 : T_1, ..., p_n : T_n$ and result of type T is mapped to:

- A layout XML file *main_activity.xml* which defines the screen using a table layout, with appropriate widgets for each of the p_i, buttons to execute *uc* and to clear the screen, and a field to display the result.
- A view controller *MainActivity.java* to read data from the screen, invoke the validator bean, display errors, call *uc* via the validator, and display its result.
- A validator *ucBean.java* which checks that the form field data items represent values of the parameter types, and converts the string values from these fields into values of the parameter types. A check is also included that each precondition of *uc* is valid for the converted values. These preconditions can express security checks (eg., that there is a logged-in user, or that input data fields do not contain SQL injection coding).
- *ModelFacade.java*, which contains the code of the *uc* as an operation $uc(T'_1\ p_1, ..., T'_n\ p_n)$ where the T'_i are the Java 8 types of the T_i. This operation returns a value of type T'. *ModelFacade* is a singleton class, and provides a single point of access to the functional core of the app. Presentation tier components relay UI-initiated requests to the *ModelFacade*. Typically it is also the delegate of callbacks from integration tier components performing asynchronous request processing, eg., Internet access.

Auxiliary Java files are also produced:

- *E.java* for any class *E* used by the app;
- *EVO.java* for each entity *E*, which is a value object class to transfer the data of *E* between app components;
- If some class is ≪*persistent*≫, a DAO called *Dbi.java* is created, to manage an SQLite database which stores the persistent class data (device support for SQLite version 3.8+ is required);
- If some class is ≪*cloud*≫, a DAO called *FirebaseDbi.java* is created, to manage a realtime database which stores the cloud class data;
- Enumerated types *T* are defined as Java enumerations in files *T.java*;
- *Uc.java* for use case *uc* is the classifier corresponding to *uc*, and contains the (static) attributes and operations of *uc*;
- *Ocl.java* is the OCL library for Java 8.
- The *FileAccessor.java* component is always provided.

The *cg/cg.cstl* file should be an appropriate code generation script for Java 8 generation when the "Android" option is selected on the *Build* menu. The *.cstl* file is used to generate functional code of *ModelFacade* and the auxiliary classes. It is possible to change the code generation strategy by editing or replacing *cg/cg.cstl*. Other generators, eg., for Kotlin or for Java 7, could be substituted [19].

If there are (public) extension use cases *extuc* of *uc*, then the input fields, buttons and result field for these are also placed on the same screen as *uc*. *extuc* can read attributes *Uc.att* of *uc*, but otherwise is independent of *uc*. The behaviour of *extuc* is defined in an operation *extuc* of *ModelFacade*. Private use cases also have corresponding operations in *ModelFacade*, but no UI components.

Figure 3 shows the general Android architecture for single-screen apps. In terms of the VIPER pattern, the screen has the View role, the *ModelFacade* the Interactor role, the *MainActivity* the Presenter role, and the entity beans and *Dbi* the Entity role. A Router is only included for multi-screen apps.

4.2 Multiple-screen apps

When there are two or more independent public use cases (including EIS use cases of the standard forms *createE*, *deleteE*, *editE*, *listE*, etc), a tabbed app is generated, with a separate tab/screen for each use case, implemented as an Android *Fragment*. The *FragmentPagerAdapter* and *ViewPager* Android classes are used to manage transitions between screens (lateral navigation) via tab selections or swipe gestures. *MainActivity.java* defines the host activity for the individual fragments. Figure 6 shows an example of the interfaces produced for such apps.

The general Android architecture for multi-screen apps is shown in Figure 4. In the case of *listE* operations, a *RecyclerView* is used to provide scrollable lists of *E* instances. This provides cell recycling for efficient display of large collections. Selection events on the list are handled by the *MainActivity* view controller, which updates the model facade accordingly (Figure 4). A *GridLayout* is used for individual list items by default, however the developer may substitute any alternative layout which supports the required widgets for items.

A share price analysis app using predefined components for Internet access, data conversion and graph display of share price data is shown in Figure 7. Components *DailyQuote_DAO* and *InternetAccessor* with *InternetCallback* are included automatically in the system specification as a result of *DailyQuote* being defined with a ≪*remote*≫ stereotype. Other components (*Date Component* and *GraphDisplay*) are manually added.

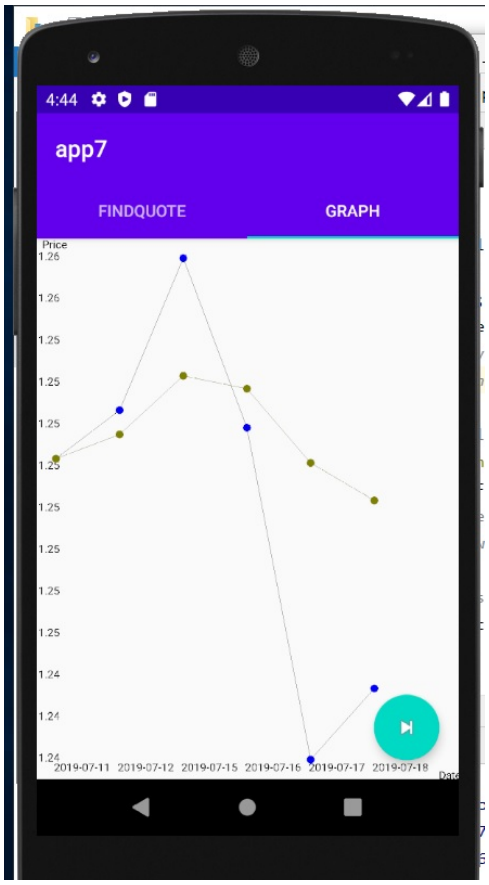


Figure 7: Interface of share analysis app

5 IOS APPS

A similar approach is adopted for iOS apps based on UIKit and Swift 4+, and for SwiftUI apps based on Swift 5. The organisation of the business and integration tiers are essentially the same as with Android.

5.1 UIKit apps

For single-screen UIKit apps the architecture of Figure 8 is used.

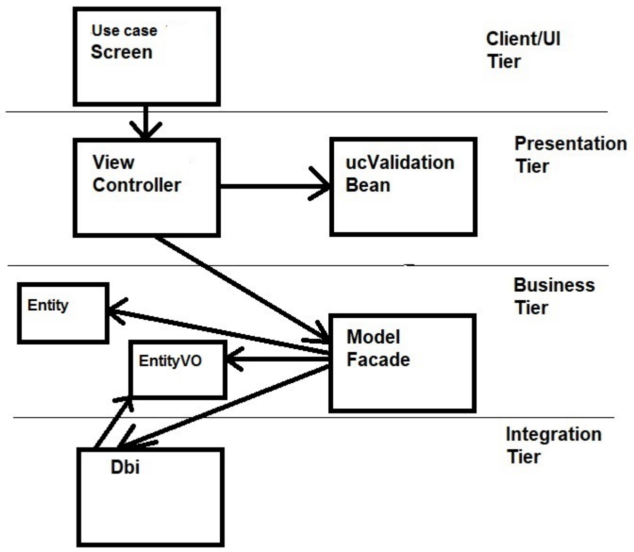


Figure 8: Architecture for single-screen UIKit iOS apps

Instead of Android Activities or Fragments, iOS *UIView Controller* components are generated to manage screens. These depend on the UIKit library. Other parts of the app, such as validation beans and model facade, are written in standard Swift 5. In this case, the code generator *cg*/*cg.cstl* should be the Swift 5 code generator specification.

As with Android, a tabbed interface is assumed for multi-screen apps. For list views, *UITableView* is used to display scrollable lists with cell recycling. The actual screen layouts for iOS/UIKit apps are not synthesised, due to the proprietary nature of the iOS XML screen description format. Instead, the developer can produce appropriate UI screens using Xcode, and connect these to the generated view controllers.

5.2 SwiftUI apps

Whilst Android and iOS/UIKit apps use an event-based UI approach, SwiftUI is based instead on the principle that the UI layouts and content should be a function of the app business data. UI screens are defined as SwiftUI View components, which can contain local data (@State variables) and observe model data (@ObservedObject or @EnvironmentObject variables). Our architectural approach is consistent with SwiftUI, with the *ModelFacade* serving as the central data source for the UI content.

Instead of a VIPER architecture, we use a MVVM approach for SwiftUI apps, with SwiftUI views taking the role of both the screens and view controllers of the UIKit approach. The *ModelFacade* has the ViewModel role as a source of observable data to which views subscribe. We also combine the roles of use case value objects and use case validators. Other tiers of the architecture remain unchanged from the UIKit approach. Figure 9 shows the architecture for multiple-screen SwiftUI apps. Figure 10 shows the bond analysis app version in SwiftUI.

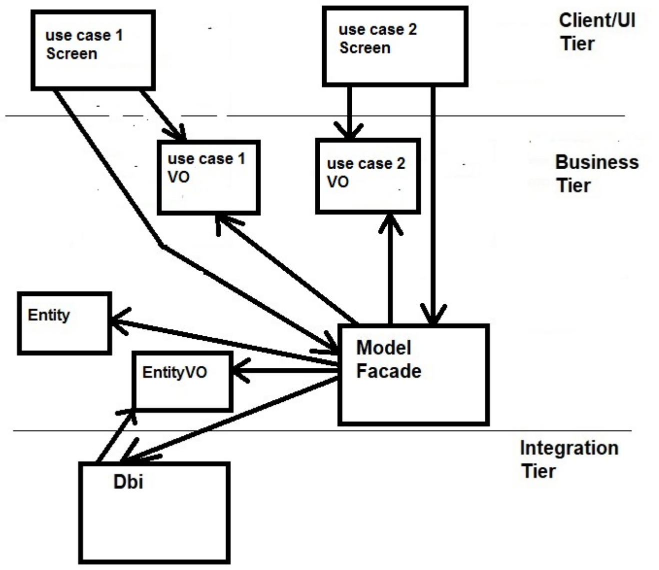
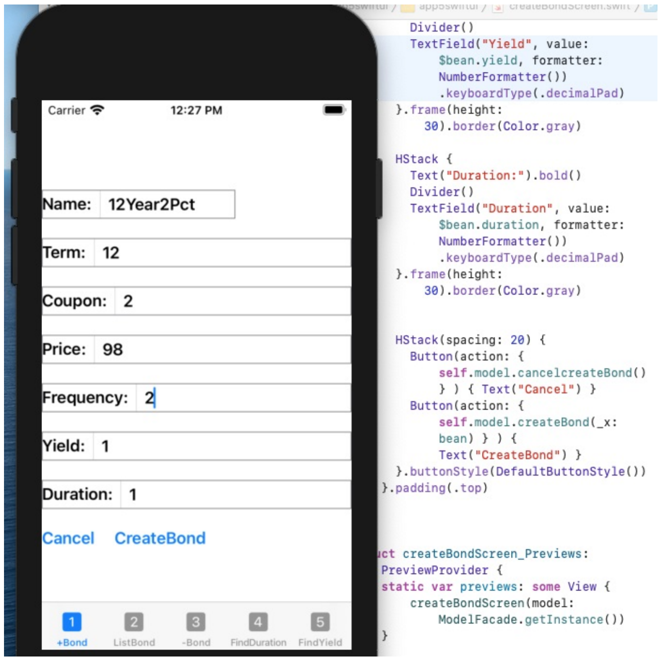


Figure 9: Architecture for multiple-screen SwiftUI iOS apps



```
    Divider()
    TextField("Yield", value:
        $bean.yield, formatter:
        NumberFormatter())
        .keyboardType(.decimalPad)
  }.frame(height:
      30).border(Color.gray)

  HStack {
    Text("Duration:").bold()
    Divider()
    TextField("Duration", value:
        $bean.duration, formatter:
        NumberFormatter())
        .keyboardType(.decimalPad)
  }.frame(height:
      30).border(Color.gray)


  HStack(spacing: 20) {
    Button(action: {
        self.model.cancelcreateBond()
        } ) { Text("Cancel") }
    Button(action: {
        self.model.createBond(_x:
        bean) } ) {
        Text("CreateBond") }
  }.buttonStyle(DefaultButtonStyle())
}.padding(.top)


ct createBondScreen_Previews:
PreviewProvider {
static var previews: some View {
    createBondScreen(model:
        ModelFacade.getInstance())
}
```

Figure 10: App 5 in SwiftUI

The authentication and realtime datastore services of Firebase also utilise MVVM, so that a SwiftUI app generated by our approach will have a consistent architecture at each tier.

6 EVALUATION

We evaluate the approach using some small but useful apps from different domains: a health and fitness advisory app; a business app for bond valuation (Figures 1, 2, 6, 10); an app for share price analysis (Figure 7), and an app for environmental impact rating of journeys. The three larger apps all involve non-trivial functionality such as numerical optimisation procedures, computation of financial indicators, and distance computations and map interaction. The specifications and code of all the examples are available at the AgileUML Github site.

We compare the size of the app specifications, the development effort, and the size of the generated app code for each app framework (Table 2). We also evaluate the number of quality flaws (duplicated code; excessive cyclomatic complexity; excessive numbers of parameters and local variables; excessive coupling; excessive class or operation size; excessive numbers of operations) in the generated code, using the flaw indicators and thresholds of [11].

For each target platform, the average ratio of the size of the generated code to the authored specification size is over 10 times, producing in principle a 90% reduction in effort of our approach compared to manual coding of the apps. The code quality is also generally higher (lower flaw density) than that of developer-coded Eclipse projects in [11]. The Eclipse projects of [11] had flaw density values ranging from 0.005 to 0.04 flaws per LOC, with an average around 0.015. Thus for these cases we can show that the approach is potentially beneficial in terms of development effort and quality.

7 COMPARISON WITH RELATED WORK

There are many approaches to app production, ranging from web app technologies and tools (eg., Sencha Touch), through hybrid apps (React Native, Xamarin) to MDE approaches targeted at native platforms [2, 4, 6, 10, 13, 26, 28, 29], of which the leading examples are MD^2 [12] and PIMAR [31]. These are both based on the definition of specialised DSLs for the mobile domain, which is the principal MDE approach for mobile development [20, 25, 30]. However, there are problems arising from the maintenance and management of DSLs, and we have decided to base our approach on a subset of standard UML, as with [15, 21].

A comparison of cross-platform approaches showed that MDE approaches can be superior in performance to other approaches, and attain a similar level of efficiency to manually-coded native apps [5].

Our approach is focussed on the primarily declarative specification of apps via UML class diagrams and operations and use cases with pre- and post-conditions. In contrast, both MD^2 and PIMAR adopt an event/action specification paradigm, which is appropriate for an MVC architecture, but not well-aligned to the SwiftUI data-centered paradigm. We have chosen a textual notation for defining detailed functionality, because text is generally faster to write and modify compared to graphical notations such as the UML activity diagrams used in PIMAR.

Table 3 compares the approach of this paper to MD^2 and PIMAR.

MD^2 and PIMAR are mainly focussed on data-driven business apps, supporting data management of persistent data. PIMAR also supports AR apps, but with the use of manual coding. We prefer to provide fully-automated code generation, requiring no manual

App case	Specification size (LOC)	Development time (hours)	Generated code LOC (Android)	Generated code LOC (iOS/UIKit)	Generated code LOC (SwiftUI)	Flaws/ LOC (Android)	Flaws/ LOC (iOS)	Flaws/ LOC (SwiftUI)
Health app	55	1	1514	903	450	0.002	0.003	0.007
Bond app	95	4	2014	1247	885	0.0025	0.005	0.0045
Share app	180	12	2279	1653	1644	0.003	0.003	0.004
Environmental app	144	8	1374	1690	1780	0.0043	0.005	0.004

Table 2: Evaluation on app cases

Aspect	AgileUML	MD^2	PIMAR
Specification Notations	Graphical or textual UML.	Textual DSL.	Textual and graphical DSL.
Scope	Produces complete functional code.	Manual coding needed for custom functionality.	Manual coding needed for custom functionality.
Code Generation	User-configurable code generation; Full OCL support	Fixed code generator; partial OCL support	Fixed code generator; partial OCL support
Domains	Finance; mHealth; general. Not games/AR	Data-driven business apps	Data-driven business apps; AR
Platforms	Android/Java 8; iOS Swift 4/5 UIKit/SwiftUI	Android/Java iOS Objective-C UIKit	Android/Java iOS Objective-C UIKit

Table 3: Comparison of mobile app synthesis approaches

coding. This eliminates the problem of duplicated coding effort for multiple platforms, and the work of integrating and maintaining generated and manual code. Instead of defining a new specification notation for mobile apps, we have reused the UML-RSDS subset of the standard UML and OCL languages. We have conceptualised app framework elements (such as view controllers, cached DAOs, cloud datastores) in terms of existing UML-RSDS elements (Table 1). This has the benefit that not only can apps for different platforms be generated from one specification, but also conventional desktop application, web service or EIS versions of the same system can be produced. The same tooling can be used to generate additional C, Swift or C++ code files for use with an app. Relevant code-generation techniques and software patterns can be directly reused for mobile apps from other application types. Configurable code generation is a particular advantage for the mobile domain, where language turnover and evolution has been rapid. Another minor advantage of our approach is that our tooling is independent of any external software such as Eclipse. Only a Java runtime is required. The tool footprint is small, under 2MB.

In comparison to the other UML-based approaches [21] and [15], while [21] offers the additional capability of specifying temporal relationships between use cases, it only appears to support Android currently, and to generate only skeleton code. The approach of [15] uses fine-grained modelling of Android components/services within UML, and detailed specification of functionality as UML activity diagrams. As noted above, text is probably more appropriate for detailed functional modelling, instead of process diagrams. We also adopt a component-based approach, but our components are specified in a high-level manner, abstracting from the details of corresponding iOS and Android services.

Other rapid development approaches include tools which convert UI sketches into UI code [3], [22]. These could be used as a means of specifying UI designs for the front end of apps. Compared to template-based or data-based app builders such as Microsoft PowerApps[5] or Google AppSheet[6], we provide greater flexibility in functional specification and control over app elements.

8 LIMITATIONS AND FUTURE WORK

One omission in our approach is the capability to define alternative visual styles or themes for app UIs. We intend to address this by defining a declarative UI description language (DSL), which could specify desired styling for visual elements at an application level or individual screen or widget level [1]. DSL specifications for elements would override the default code generation choice for the elements (eg., that GridLayout is used for object list displays in Android). Thus they would be relatively concise.

We do not consider game apps, which require specialised techniques and elaborate UI mechanisms. An MDE approach for game apps is investigated by [7].

The generators and sets of predefined components will be extended to include more mobile services and the handling of events/data from specialised sensors. Further optimisation and security strategies will be applied, for example, enabling use cases with an ≪*asynchronous*≫ stereotype to be implemented asynchronously, in order to reduce load on the main thread. Components for machine learning algorithms and NLP will be added to support m-health app specification.

[5]https://powerapps.microsoft.com
[6]https://www.appsheet.com

While UML and OCL have the advantage of being internationally-standardised and widely-used languages, they also have limitations. In particular, OCL lacks both map types and function types, which are extensively used in typical Android and iOS apps (eg., JSON data is represented as maps in iOS, and Java lambda expressions and Swift closures are used to facilitate callbacks from asynchronous processes to their initiators). We are in the process of extending OCL to include map and function types [18].

Currently we do not formally track specification changes to enable selective update of generated app code. However the modular nature of the code generation process enables code changes to be limited. For example, modification of one use case activity will only result in changes to its own view controller and validator bean code and to the $ModelFacade$.

In the long term we aim to derive mobile app data models and UI specifications from natural language requirements and sketches. This could further accelerate app development and reduce the effort expended by developers. We also aim to provide a design assistant to guide developers in the choice of app components and organisation.

9 BROADER IMPACTS

Short time-to-market and high usability are priorities for mobile applications. In addition, apps should conform to the architectures supported by the mobile platforms, avoid security problems, and avoid inefficient or bloated code.

Agile MDE development of native mobile applications can avoid the cost of manually writing code for the same app for multiple device platforms/versions, and can enable correct architectures, security policies and high-quality coding practices to be automatically imposed.

While increased automation of software production may cause reduced employment opportunities for software developers, this is likely to be counterbalanced by continuing demand for new and more advanced software applications.

CONCLUSIONS

We have described the AgileUML approach to synthesis of mobile apps, which is a lightweight MDE approach emphasising the production of complete functional apps based on minimal coding in high-level specifications. In contrast to other MDE approaches for mobile apps, we provide a mapping from declarative UML models to app elements. The effectiveness of the approach has been shown by applying it for practical app development.

REFERENCES

[1] L. Alwakeel, K. Lano, *Model-driven development of mobile applications*, Doctoral Symposium, ECOOP 2020.

[2] H. Behrens, *MDSD for the iPhone*, OOPSLA Companion proceedings, 2010.

[3] T. Beltramelli, *pix2code: Generating code from a GUI screenshot*, CoRR abs/1705.07962, 2017.

[4] C. Bernaschina, S. Comai and P. Fraternali, *IFMLEdit.org: Model Driven Rapid Prototyping of Mobile Apps*, 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 207–208.

[5] A. Bjorn-Hansen, C. Rieger, T. Gronli, et al., *An empirical investigation of performance overhead in cross-platform mobile development frameworks*, Empirical Software Engineering, vol. 25, pp. 2997–3040, 2020.

[6] L. Brunschwig, E. Guerra, J. de Lara, *Towards access control for collaborative modelling apps*, LowCode@MoDELS, 2020

[7] M. Derakhshandi, S. Kolahdouz-Rahimi, J. Troya, K. Lano, *Using model-driven approach for developing Android-based multi-player games*, Submitted to ASE, 2020.

[8] Eclipse Agile UML project, https://projects.eclipse.org/projects/ modeling.agileuml, 2020.

[9] M. Fowler, *Patterns of Enterprise Application Architectures*, Pearson Education, 2003.

[10] R. Francese, M. Risi, G. Scanniello, G. Tortora, *Model-Driven Development for Multi-platform Mobile Applications*. In: Abrahamsson P., Corral L., Oivo M., Russo B. (eds) Product-Focused Software Process Improvement. PROFES 2015. Lecture Notes in Computer Science.

[11] X. He, P. Avgeriou, P. Liang, Z. Li, *Technical debt in MDE: A case study on GMF/EMF-based projects*, MODELS 2016.

[12] H. Heitkotter, T. Majchrzak, H. Kuchen, *Cross-platform MDD of mobile applications with MD^2*, SAC 2013, ACM Press, 2013.

[13] J. Grundy, M. Abdelrazek, M. Kissoon, *Vision: improved development of mobile eHealth applications*, MOBILESoft 2018, ACM, 2018.

[14] F. Jouault, J. Bezivin, *KM3: a DSL for metamodel specification*, ATLAS team, INRIA, 2006.

[15] F. A. Kraemer, *Engineering Android Applications Based on UML Activities*, MODELS 2011. Lecture Notes in Computer Science, vol 6981, 2011.

[16] K. Lano, *A compositional semantics of UML-RSDS*, SoSyM, vol 8, 2009, pp. 85–116.

[17] K. Lano, *Agile Model-driven Development with UML-RSDS*, Taylor and Francis, 2016.

[18] K. Lano, S. Kolahdouz-Rahimi, *Extending OCL with map and function types*, FSEN 2021.

[19] K. Lano, Q. Xue, S. Kolahdouz-Rahimi, *Agile Specification of Code Generators for Model-Driven Engineering*, ICSEA 2020.

[20] O. Le Goaer and S. Waltham. *Yet another DSL for cross-platform mobile development.* In Proceedings of the First Workshop on the Globalization of Domain Specific Languages (GlobalDSL '13), 2013.

[21] V. Lopez-Jaquero, P. Gonzalez, F. Montero, and J. Pascual Molina, *UML2App: Towards the automatic generation of user interfaces for mobile devices*, In Proceedings of the XX International Conference on Human Computer Interaction (Interaccion '19), 2019.

[22] Microsoft, *sketch2code*, https://sketch2code.azurewebsites.net, 2020.

[23] OMG, *Semantics of a Foundational Subset for Executable UML Models*, https://www.omg.org/spec/FUML/About-FUML/, 2018.

[24] K. Pernice, R. Budiu, *Hamburger menus and hidden havigation hurt UX metrics*, www.nngroup.com/articles/hamburger-menus, 2016.

[25] A. Ribeiro and A. Rodrigues da Silva, *XIS-mobile: a DSL for mobile applications.* In Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14). pp 1316–1323, 2014.

[26] C. Rieger et al. *A model-driven approach to cross-platform development of accessible business apps.* Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20). pp. 984–993, 2020.

[27] J. Ruiz et al., *Evaluating UI generation approaches: model-based versus MDD*, SoSyM, 2019.

[28] M. Sharbaf, *A Framework for automatic generation of location-based Android applications*, https://mdse.ui.ac.ir/project/alba, 2020.

[29] E. Umuhoza, M. Brambilla, *Model Driven Development Approaches for Mobile Applications: A Survey.* In: Younas M., Awan I., Kryvinska N., Strauss C., Thanh D. (eds) Mobile Web and Intelligent Information Systems. MobiWIS 2016. Lecture Notes in Computer Science, vol 9847, 2016.

[30] D. Vaquero-Melchor, et al., *Active domain-specific languages: Making every mobile user a modeller*, MODELS '2017, pp. 75–82, 2017.

[31] S. Vaupel, G. Taentzer, R. Gerlach, M. Guckert, *Model-driven development of mobile applications*, Sosym, Feb. 2018.