A Process for Integrating Agile Software Development and Model-Driven Development

Hessa Alfraihi
dept. of informatics
King's College London
London, UK
hessa.alfraihi@kcl.ac.uk

Kevin Lano
dept. of informatics
King's College London
London, UK
kevin.lano@kcl.ac.uk

***Abstract*—Agile software development and Model-Driven Development (MDD) are two software engineering paradigms that contribute to enabling the rapid development of applications. Previous approaches have proposed the integration of Agile and MDD, however these approaches are either specific to one application domain, or fail to cover the complete development cycle, for example, to include requirements engineering. To address this problem we propose a general and comprehensive process that integrates Agile development and MDD, and that allows applications to be safely developed in an iterative and incremental manner. We also report on a case study to evaluate the application of the proposed process.**

***Index Terms*—agile development, model-driven development, agile model-driven development, process**

I. Introduction

Agile development methods have gained increasing attention among the software development community [6]. They are centred around some values, practices and principles proposed by the Agile Manifesto [3]. These values are related to a tight communication with the customer, the frequent delivery of the application, response to changes in requirements, and so forth. Model-Driven Development (MDD) [14] is another software development paradigm that is distinguished from other approaches by the fact that models are not only used for analysis and design aspects, but also for producing artifacts semi-automatically.

Although researchers are already trying to integrate both Agile development and MDD, this integration has not been explored deeply so far [1], [5]. The goal of this paper is to address this problem by proposing a process that integrates both approaches effectively. Then, we evaluate the feasibility of the proposed process by a case study.

The remainder of this paper is structured as follows: in Sect. II we discuss the related work while in Sect III, we present the proposed process, outlining its main phases and activities. Sect. IV reports the application of the proposed process to develop a code generator. In Sect. V, we evaluate the feasibility of the proposed process. Finally, in Sect VI we conclude the paper and discuss our future work.

II. Related Work

Integrating Agile development and MDD (Agile MDD) is a new discipline and research in this area just started [1], [5]. However, there exist some approaches that attempt to integrate Agile and MDD in the areas of finance, web applications, and telecommunications.

Zhang and Patel in [7] apply Agile MDD process to develop a project for telecommunication by integrating two existing processes: System-Level Agile Process (SLAP) and MDD process. SLAP is Scrum-based process that contains three phases: requirements and architecture, development, and system integration feature testing (SIFT). On the other hand, MDD process follows a V-model process which includes the following activities: application requirements specification and architecture, requirements analysis and high-level design, detailed design, code generation and UML unit and integration testing, subsystem testing and system testing. In order to achieve Agile MDD they establish a simple correspondences between MDD activities and SLAP sprints to end up with three phases: application requirements and architecture, development, SIFT.

Kulkarni *et al* [9] propose a method for integrating Agile method and MDD. They argue that Agile development contradicts with MDD as some activities are not suitable to be done during short iterations. Therefore, they introduce "meta-sprint" in addition to normal sprints in their Scrum process. This meta-sprint is dedicated for tasks that require longer duration such as detailed exploration or research.

Guta [8] proposes MDD process that follows a traditional iterative and incremental process. The development process is based on parallel collaboration between three different teams namely; Agile development team, business analyst team, and MDD team. It encompasses initial phase and development phase. Initial phase contains domain model extraction, architecture elaboration, and code generator set up. On the other hand, development phase is an iterative process where the domain model is extended and the other artifacts are generated. In this approach, MDD process is only applied on a subset of the system while the other features are hand-crafted.

Besides that these processes are platform-specific, they only describe the general course of the process and they don't consider the full development lifecycle. We propose a more general process that considers the full development lifecycle activities which can be applied to different domains.

III. An Agile Model-Driven Development Process

In order to develop the process stages, we reviewed the literature to identify what are the main practices and activities in MDD processes. To this end, we adopted some MDD stages from [11] and [17] . Similarly, to integrate Agile practices, we analysed three different Agile methods: Scrum [13], Extreme Programming (XP) [4], and Agile modelling [2]. Generally, we used Scrum as the backbone process since it provides a management framework for managing and controlling iterative work at the project level. Some other practices such as pair programming, refactoring, active stakeholder participation and architecture envisioning were utilised from XP and Agile modelling. In our Agile MDD, we adopted the three phases from [17] and the parallel tooling sub-team from [8].
The activities of our process are assigned to a limited number of roles, and one developer can play multiple roles. We identified the following role and responsibilities:

- **Product owner:** is responsible for creating and maintaining the product backlog.
- **Modeller:** is responsible for developing different components of the system.
- **Tester**: is responsible for testing at model and implementation level.
- **Specification library manager:** is responsible for managing and maintaining the specification library.
- **Tooling sub-team:** is responsible for providing extension or adaptation to the tool.

The development process starts with the *Initialisation* phase and finishes with the *Deployment* phase. Throughout the development, it follows an iterative and incremental approach. The proposed process is not tied to a particular formalism, however we propose to adopt OMG standards, i.e., SysML and UML, to be open to multiple tools and promote the interoperability of the models. An overview of the process is presented in Fig. 1 while the different phases and its stages are explained in the following subsections.

A. *Phase 0: Initialisation*

The main objective of this phase is to capture the initial information about the system such as its scope, size, environment conditions and so on. At this stage, strong collaboration with the customer is crucial to gather the required information. Furthermore, the initial requirements of the system and the architecture at a high-level scale are identified. To meet agility, it is necessary to avoid capturing too complex information since detailed information will be evolved incrementally. All the requirements are prioritised and recorded in the product backlog by the product owner as a form of user stories. Based on that, the overall release plan will be established which distributes the product backlog tasks into iterations in order to predict when the product can be released. Moreover, the tools, modelling language and platform should be identified and agreed upon at this stage.

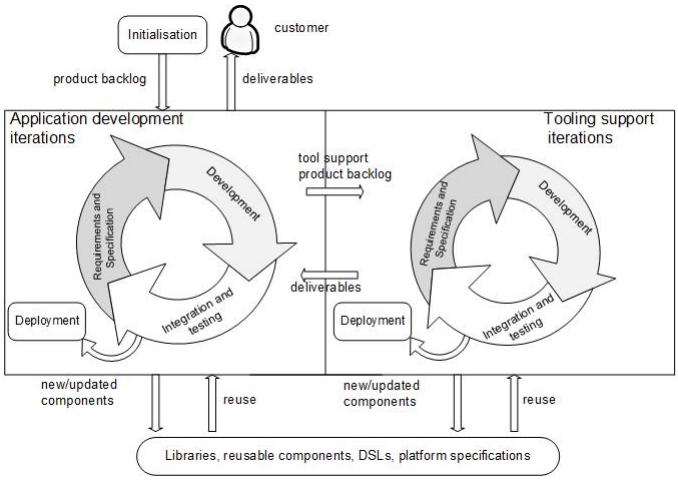

Fig. 1: An Overview of Agile MDD Process

After the initialisation phase, the development process follows an iterative cycle. This means that the development process goes through repeated phases until the system meets the customer's needs. The iterative cycle encompasses the following phases: *Requirements and Specifications*, *Development*, *Integration and Testing*.

Each iteration begins with an iteration planning activity to agree on the work to be accomplished in the upcoming iteration. The initial release plan performed in the initialisation phase can be used to feed directly the iteration planning. To support the agile philosophy, changes in requirements should be welcomed at any time. Therefore, any changes should be discussed and the product backlog is revised at this planning. Afterwards, the product owner agrees with the development team on the requirements to be implemented at this iteration according to their priorities. In order to communicate and manage the requirements effectively, user stories are produced to describe the requirements in one or two sentences. The requirements selected to be accomplished in the current iteration are added to the iteration backlog.

Throughout the iteration, daily stand-up meeting can be conducted in order to solve the problems encountered by the developers and to allow them to catch-up with each other's progress. The daily meeting should be as short as possible, normally it lasts no more than 15 minutes.

B. *Phase 1: Requirements and Specifications*

The main objective of this phase is to analyse and specify the requirements of the current iteration. This phase includes two main stages: Requirements Analysis and Requirement Specifications.

1.1 Requirements Analysis

In this stage, the requirements of the current iteration are defined and structured in more detail. Both functional and non-functional requirements should be identified clearly. Since, non-functional requirements are ill-defined in Agile development [12], more attention should be paid to identifying

them early at the development stage. Some requirements mechanisms such as prototypes and scenario analysis can be used to agree and clarify the meaning of requirements with the customer. During this stage, any existing components that can be used to support new requirements should be identified. Furthermore, any components which have potential for future reuse should be identified where possible. The recognition of reusable components within a product line may result in the establishment of a substantial library of components to support the development of related products.

1.2 Requirements Specification
Once the requirements are identified, modellers start developing models that fulfil these requirements. The initial architecture should be revised and amended accordingly. In the context of MDD, metamodels and transformations are considered the main elements of the process and any models should conform to metamodels. As with any artifact, developers can reuse metamodel artifacts provided by standards such as UML and MOF or developed in previous common projects. Any identified reusable components should be imported from and/or added to the library by the specification library manager.

When the representation of the metamodels is not available, developers need to define it to capture the abstract syntax and constructs used to define models. Transformations should also be identified, these include transformations between different models or from models to text. Afterwards, models are created that specify the requirements precisely. Different perspectives of the system are identified to be modelled and a set of models is selected accordingly e.g. use case, class diagram, activity diagram. Pair modelling, analogue to XP's pair programming [4], can be followed during modelling where two developers work together. This should help detecting and resolving modelling issues instantly [7]. At this stage, any technical specification should be avoided.

C. *Phase 2: Development*

The main objective of the development phase is to produce a technical specification of the system and to produce an executable system that fulfils the functional and non-functional requirements. This phase includes two main stages: Design and Implementation.

2.1 Design
The developers model the detailed structure and behaviour of the system that fulfils the functional and non-functional requirements. Although extensive customer involvement is a key factor in Agile development, in our process the customer is selectively involved in tasks such as requirements capturing and reviewing. This is due to the facts that most customers don't have required knowledge of MDD activities such as modelling, designing and implementation. Following the Agile practice *incremental design*, the developers design and specify the platform-specific aspects by refining the platform-independent models which have been defined in the previous stage. These specifications describe the technical specification of the target platform (e.g., EJB, .NET) to allow generating the code and transformation can be employed to refine some artifacts. Formal verification at the specification level can be used to ensure the key properties are preserved from one iteration to the next. Following the Agile practice *refactoring*, developers should review the specification structure continuously to ensure the best design of the system. Since models are the intrinsic driver in development process, when change occurs or defects are discovered, the resolutions for the issues should be at the modelling level. Before and after every change, all test cases should pass.

2.2 Implementation
At this stage, developers produce the software that is testable and executable. MDD differs from other software development approaches in that the code is partially or completely generated from the models. The level of automation that translates models into code can be varied from partial to complete implementation of the system. Although the code is generated from models, it is still necessary to run tests against the code to make sure the model semantics are as expected. When some part of the system cannot be generated automatically, it can be written and manually added and tested.

In parallel to the software development process, a tooling sub-team can be employed to provide technical support such as necessary extensions or adaptations of tools. Having in-house team is found to be effective to help when issues arise [16]. The same proposed Agile MDD process can be used by the tooling team considering the development team as the customer who issues requirements for tools. The sub-team is optionally employed when needed.

D. *Phase 3: Integration and Testing*

The main objective of this phase is to integrate the developed parts of the system and to make sure they behave as expected. This phase includes two main stages: Integration and Testing.

3.1 Integration
Since one of the intrinsic characteristic of agile development is to develop the system in an incremental fashion, the increments need to be integrated and tested frequently. Therefore, by adopting the agile practice of *continuous integration*, developers need to integrate their work as soon as it is completed.

3.2 Testing
Integrated parts such as components and subsystems are verified by regression tests to detect errors early in the development. Although testing is integrated frequently in all the phases of the iteration, testing should be performed at the end of the development process as well [15]. In the context of MDD, automatic testing (model-based testing) can be carried

out. However, manual testing according to various levels of testing activities can be used.

At the end of the iteration, an increment of the system is released and delivered to the customer for assessment and review. When some requirements cannot be completed during the iteration, they are carried over to the next one. To assess the development process and assess what has been done during the iteration, a review meeting is held.

E. *Phase 4: Deployment*

When the system has been fully implemented and reaches a stable version, the system will then be deployed to the customer. The goal of the deployment is to release the system and make it ready to use by the customer without developer's supervision. This activity takes place only once at the end of the development process.

IV. Case Study: UML to C code Generator

In order to illustrate the proposed process, we applied it to the development of code generator for mapping UML to ANSI C for the UML-RSDS [10]. UML-RSDS is a MDD tool that uses a subset of UML as an input language. At the specification level, UML-RSDS models application by UML 2 class diagrams and use cases. Optionally state machines and interactions can be used. At the design level, UML activities using pseudo-code notations are used. For this case study, the stakeholders are: customers who needed to develop this system, the UML-RSDS development team, and end users who use such a system.

A. Phase 0: Initialisation

Besides the scope and the size of the system, the initial requirements were identified. The main functional requirement for the generator is *(F1): Translate UML-RSDS designs (UML class diagram, use cases, OCL, activities) into ANSI C code*. Using goal decomposition, F1 is decomposed into five sub-goals:

- F1.1: Translation of types
- F1.2: Translation of class diagrams
- F1.3: Translation of OCL expressions
- F1.4: Translation of activities
- F1.5: Translation of use cases

Furthermore, non-functional requirements were identified such as:

- NF1: Termination, given correct input
- NF2: Efficiency: input models with 100 classes and 100 attributes per class should be processed within 30 seconds
- NF3: Modularity of the transformation

The dependencies and priorities of the requirements were identified and consequently the product backlog was created. Furthermore, the initial architecture was identified as a sequential decomposition of model-to-model transformation (*design2C*) and model-to-text transformation (*genCText*) (Fig. 2). In the release plan, the product owner organised the development into five iterations, each of which develops one translation. Each iteration was given a maximum duration of one month. In the following subsections, we present the iterations. Due to limited space, only the first three iterations are explained in detail.

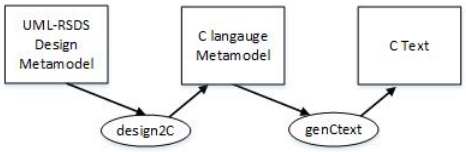

Fig. 2: C Code Generator Architecture

TABLE I: User Stories of Informal Scenarios for types2C.

ID	User Story
F1.1.1.1	Translate String type to char*
F1.1.1.2	Translate int, long, double types to same-named C types
F1.1.1.3	Translate boolean type to unsigned char
F1.1.2	Translate Enumeration type to C enum
F1.1.3	Translate Entity type E to struct E* type
F1.1.4.1	Translate Set(E) type to struct E** (array of E', without duplicates)
F1.1.4.2	Translate Sequence(E) type to struct E** (array of E', possibly with duplicates)

B. Iteration 1: Translation of Types

This iteration concerns the mapping of data types such as boolean, int, long, etc. into C representations by defining the sub-transformation *types2C*. During the iteration planning, the development team planned to implement *F1.1: Translation of types*.

Phase 1: Requirements and Specification

Detailed requirements elicitation identified the following requirements: (i) the source language, (ii) the target language, (iii) the mapping requirements, (iv) non-functional requirements NF1, NF2, NF3. The main user story for F1.1 was:

As a developer, I want to translate a UML type to its corresponding C type.

This user story was decomposed into further sub-user stories and their scenarios are presented in Table I.

The requirements specification formalised these translations as UML-RSDS rules, defining the post-conditions of a transformation *types2C*. The corresponding part of *genCtext* was developed alongside *types2C*.

Phase 2: Development

The source language for this transformation was identified as

the *Type* class and its sub-classes in the standard UML-RSDS class diagram metamodel, while the initial target language is a simplified version of the abstract syntax of C programs, sufficient to represent UML types. The design in UML-RSDS is produced automatically from the formal specification. The formal specification for F1.1.1.1 and F1.1.1.2 are:

```
PrimitiveType::
name = "int" =>  CPrimitiveType->
exists( p | p.ctypeId= typeId & p.name= "int")

PrimitiveType:: name= "String" =>
CPointerType->exists(t | t.ctypeId= typeId &
CPrimitiveType->exists(p | p.name= "char" &
 t.pointsTo = p ))
```

Phase 3: Integration and Testing
Test cases were manually developed to test the functionality, in addition to inspection and formal arguments for the satisfaction of the requirements. Since this was the first iteration, integration was not needed at this stage.

C. Iteration 2: Translation of Class Diagram

The goal of this iteration was to define the sub-transformation (*classdiagram2C*) that is concerned with mapping UML class diagram into C. During the iteration planning, given its high priority, it has been planned to implement *F1.2: Translation of class diagram* at this iteration.

Phase 1: Requirements and Specification
During the requirement phase, the class diagram elements *Entity*, *Operation*, *Property*, and *Generalisation* were identified as the input language. Exploratory prototyping was used for further requirements elicitation.

Similar to type translation iteration, user stories for the mapping of UML class diagrams to C were informally identified.

Phase 2: Development
In order to translate class diagram classes, the developers design the following function to create C code getters for properties:

```
CMember:: query getterOp(ent : String): String
post: result = type + " get" + ent + "_" +
name + "(struct " + ent + "* self) { return
self->" + name + "; }\n"
```

Similar functions were defined to express other aspects of UML classes in C.

Phase 3: Integration and Testing Phase
Testing, inspection, and formal arguments were used for validating and verifying the requirements. Both iteration 1 and 2 have been integrated and grouped as one executable jar file. Integration testing was conducted to make sure the integration behaved correctly.

D. Iteration 3: Translation of Expressions

This iteration concerns the mapping of OCL expressions to C. During the iteration planning, the development team planned to implement *F1.3: Translation of OCL expressions*. There are many cases to consider with OCL expressions mapping, thus this functionality was divided into four subcategories: (i) translation of basic expressions, (ii) translation of logical expressions, (iii) translation of comparator, numeric and string expressions, and (iv) translation of collection expressions. Due to its large size, it was not possible to complete it in one month.

Phase 1: Requirements and Specification
Detailed requirements elicitation identified the mapping scenarios of the above mentioned subgroups of OCL expressions. For brevity, only the mapping scenarios for the basic expressions are presented in Table II.

TABLE II: User Stories of Informal Scenarios for Basic Expressions.

ID	User Story
F1.3.1.1	Translate self in OCL expression to self as an operation parameter in C representation
F1.3.1.2	Translate variable v or v[ind] to v or v[ind'-1] in C representation
F1.3.1.3	Translate data feature f of context E with no objectRef to self → f (E is root) or getE_f(self) (otherwise)
F1.3.1.4	Translate operation call op(e1,...,en) or obj.op(e1,...,en) of instance entity scope op of E to
F1.3.1.5	Translate call op(e1,...en) of static/application scope op to op(e1',...,en')
F1.3.1.6	Translate col[ind] ordered collection col to (col')[ind'-1]
F1.3.1.7	Translate E.allInstances to e-instances
F1.3.1.8	Translate value of enumerated type, numeric or string value to value
F1.3.1.9	Translate boolean true, false to TRUE,FALSE

Phase 2: Development
during design activity, for each category of expressions the developers modelled the detailed specifications, for example:

```
BasicExpression::
query mapBasicExpression(ob :
Set(CExpression),aind : Set(CExpression),
pars : Sequence(CExpression)) : CExpression
pre:
  ob= CExpression[objectRef.expId] &
  aind= CExpression[arrayIndex.expId] &
  pars= CExpression[parameters.expId]
post:
  (umlKind= value =>
  result= mapValueExpression(ob,aind,pars))
  & (umlKind = variable =>
  result= mapVariableExpression(ob,aind,pars))
  & (umlKind = attribute =>
  result= mapAttributeExpression(ob,aind,pars))
  & (umlKind = role =>
  resul= mapRoleExpression(ob,aind,pars)) &
  (umlKind = operation =>
  result= mapOperationExpression(ob,aind,pars))
  & (umlKind = classid =>
  result= mapClassExpression(ob,aind,pars)) &
  (umlKind= function =>
  result= mapFunctionExpression(ob,aind,pars))
```

Phase 3: Integration and Testing
Testing and inspection were used for validation and verification. Testing of these operations revealed some errors regarding the meta-models such as error in the value of the multiplicity. The generation

TABLE III: Development effort for UML-RSDS code generators (person-month).

	Java 4	Java 7	C#	C++	C
Requirements Analysis	6	2	3	6	4.5
Implementation	12	4	4	6	1
Testing	6	1	1	2	0.5
Maintenance	6	1	1	3	0
Total	30	8	9	17	6

of model.txt by the UML-RSDS tools needed to be adjusted in several cases to ensure that appropriate information was available for UML2C. It was decided that the integration of iteration 3 will be conducted later with iterations 4 and 5.

Translation of activities, and use case were implemented in iterations 4 and 5 respectively.

E. Phase 4: Deployment

The code generator was deployed and delivered as Java jar executables: iteration 1 and 2 were grouped in one executable (uml2Ca.jar) and iterations 3, 4, and 5 in (uml2Cb.jar). The complete translator was referred to as (UML2C) and it can be found via the link in the footnote [1] .

V. Evaluation

Several code generators have been developed previously for UML-RSDS using Agile development with manual coding in Java (i.e, non-MDD approach) such as Java 4, Java 7, C#, and C++. Table III shows the approximate effort in person-months expended for each of these to date including our C code generator. An overall development effort reduction has been noticed for the C code generator using Agile MDD process. The benefits of this effort reduction are mainly due to the use of specifications instead of code, and to the use of executable modelling. The decomposition of the transformation into semi-independent phases formed a natural basis for the definition of the top-level work items in the product backlog, and this in turn led to the definition of iterations. The use of short iterations and backlogs enabled incremental development of the translator and helped organise the development. Partial specifications were defined for each separate iteration of the system, and incrementally refined. Refactoring was extensively used to improve the specification, in particular the removal of duplicated functionality. The agile emphasis on simplicity helped to reduce the specification complexity: in particular only class diagrams and OCL constraints are used to define the translator, without activities or other UML models. Also, we found it effective in handling changing requirements to translate static operations. Overall, we consider the combination of agile and MDD to be effective in improving system quality and reducing development time. Further techniques, such as model-based testing and pair modelling, will be investigated in future work.

VI. Conclusions and Future Work

In this paper, we presented a systematic process for integrating Agile development and MDD, describing its stages and activities. To illustrate the process, the development of a UML to C code generator was conducted and evaluated. In future work we will target evaluating the process more comprehensively. To this end, four case studies, developing the same system, will be conducted by four different teams implementing different approaches: "traditional approach" (i.e, a hand-coded non-agile approach); "MDD only" approach; "Agile only" approach; and "Agile MDD" approach. Comparing the results of these case studies should provide a clear understanding of the impact of integrating Agile and MDD on the development process.

[1] https://nms.kcl.ac.uk/kevin.lano/uml2web/

References

[1] Hessa Alfraihi and Kevin Lano. The integration of agile development and model driven development: A systematic literature review. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, 2017.

[2] Scott Ambler. *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, 2002.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, and Jeff Sutherland. The Agile Manifesto. Technical report, The Agile Alliance, 2001.

[4] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[5] Håkan Burden, Sebastian Hansson, and Yu Zhao. How MAD are we? Empirical Evidence for Model-driven Agile Development. In *Proceedings of XM 2014, 3rd Extreme Modeling Workshop*, volume 1239, pages 2–11, Valencia, Spain, September 2014. CEUR.

[6] Widia Resti Fitriani, Puji Rahayu, and Dana Indra Sensuse. Challenges in agile software development: A systematic literature review. In *Advanced Computer Science and Information Systems (ICACSIS), 2016 International Conference on*, pages 155–164. IEEE, 2016.

[7] Gábor Guta, Wolfgang Schreiner, and Dirk Draheim. A lightweight mdsd process applied in small projects. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 255–258. IEEE, 2009.

[8] Gábor Guta, Wolfgang Schreiner, and Dirk Draheim. A lightweight mdsd process applied in small projects. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 255–258. IEEE, 2009.

[9] Vinay Kulkarni, Souvik Barat, and Uday Ramteerthkar. Early experience with agile methodology in a model-driven approach. In *Model Driven Engineering Languages and Systems*, pages 578–590. Springer, 2011.

[10] Kevin Lano. Uml-reactive system design support. Technical report, King's College London, 2012.

[11] Xabier Larrucea, Ana Belen García Díez, and Jason Xabier Mansell. Practical model driven development process. *Computer Science at Kent*, page 99, 2004.

[12] Balasubramaniam Ramesh, Lan Cao, and Richard Baskerville. Agile requirements engineering practices and challenges: an empirical study. *Information Systems Journal*, 20(5):449–480, 2010.

[13] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Agile Software Development. Prentice Hall, 2002.

[14] Bran Selic. Model-driven development: Its essence and opportunities. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 7–pp. IEEE, 2006.

[15] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[16] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. A taxonomy of tool-related issues affecting the adoption of model-driven engineering. *Software & Systems Modeling*, 16(2):313–331, 2017.

[17] Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE software*, 28(2):84, 2011.