The Impact of Integrating Agile Software Development and Model-Driven Development: A Comparative Case Study

H. Alfraihi[1,3], K. Lano[1], S. Kolahdouz-Rahimi[2], M. Sharbaf[2], and H. Haughton[1]
{hessa.alfraihi,kevin.lano}@kcl.ac.uk
{sh.rahimi,m.sharbaf}@eng.ui.ac.ir

[1] Dept. of Informatics, King's College London, London, UK
[2] Dept. of Software Engineering, University of Isfahan, Isfahan, Iran
[3] Dept. of Information Systems, Princess Nourah bin Abdulrahman Univeristy, Riyadh, Saudi Arabia

Abstract. Agile and Model-Driven Development integration (Agile MDD) is of significant interest to researchers who want to leverage the best of both worlds. Currently, there is no clear evidence or proof for the real impact of such integration. As a first step in this direction, this paper reports an empirical investigation on the impact of integrating Agile and Model-Driven Development on the quality of software systems. To this end, we developed a financial application using Agile MDD, which is further contrasted with three other independent versions of the same application developed using different approaches: Agile method, MDD method, and traditional (manually-coded) method, respectively. We also compared the functionality of the systems and a variety of technical debt metrics measuring the quality of the code and its design. Based on the case study results, we have found that the use of Agile MDD shows some improvements in the product quality and efficiency.

Keywords: Agile development, Model-driven development, Agile model-driven development integration, Case study, Financial applications

1 Introduction

Agile development processes evolved to overcome some of the perceived limitations of the bureaucratic plan-driven approaches [8]. They attempt to be as lightweight as possible in terms of the development process: their primary goal is to deliver a system to the customer that meets his needs, in the shortest possible time, taking account of changes in requirements. This is achieved by having short iterations and developing software incrementally; coping with changes through specific technical practices, and by focusing on customer involvement throughout the development. A variety of agile processes exist that share the same values

and principles. The most widely used are Scrum [22] and Extreme Programming (XP) [5].

Model-driven Development (MDD) is another software development approach that has been gaining considerable attention during the last decade [3,17]. Unlike Agile development, MDD provides a capability for a high degree of rigour and formalisation by proposing a paradigm shift from code level to model level. The main aim of MDD is to separate the business logic from its implementation. This allows the developers to focus on solving the problem instead of focusing on its implementation. Particularly, MDD uses models as the primary artifact of the software development and the implementation is generated automatically (or semi-automatically) from the models. MDD promotes faster development with fewer bugs (in principle) by automatic generation of code, easier communication among stakeholders by increasing abstraction in the design, improving maintainability, and generating platform-independent solutions [15].

Agile and Model-driven Development integration (Agile MDD) is of significant interest to researchers who want to utilise the best of both worlds. However, not enough research has been carried out to investigate the impact of their integration [1,6]. More specifically, not much is known about the quality of software developed using an Agile MDD approach. The aim of this paper is to assess the impact of integrating Agile development and MDD through a case study. To that end, a comparison between four independent developments of the same application was performed, with the developments using different approaches: Agile MDD, MDD, Agile, and non-Agile hand-coded development.

The remainder of this paper is structured as follows. After a brief discussion of related work in Section 2 and our research methodology in Section 3, we describe our Agile MDD process in Section 4. In Section 5, we report on the case study, while the results are presented in Section 6. Section 7 discusses the results followed by listing the limitations of this study. Finally, Section 8 discusses the conclusion of the study and highlights areas for future work.

2 Related Work

In our previous work [2] we compare a UML to C code generator developed using Agile MDD, to previously developed code generators (for UML to Java, C++, and C#). These generators were developed using an Agile method with manual coding in Java. The results show a 33% increase in the developer's productivity and a 4 times reduction in size (LOC) for Agile MDD. Likewise, Zhang and Patel [27] compare some components developed using Agile MDD to other hand-coded components developed using Agile method. They noticed a threefold increase in productivity and higher quality in terms of reduced defects density. However, they did not show clear metrics or explain how the comparison was performed.

To the best of our knowledge, there is no case study on Agile MDD that compares the *same* software product in three ways, to one developed using MDD without Agile, to another using Agile development without MDD, and to one

developed manually in a traditional way. This study should fill in this gap and provide such comparison.

3 Research Methodology

The high-level goal of our research is to *evaluate the impact of integrating Agile development and MDD*. The research goal was refined into detailed properties, and specific measures for these properties were selected, according to the Goal Question Metric (GQM) methodology [4]. The goal leads to the following research questions:
RQ1: What is the impact of integrating Agile development and MDD on the software product?
RQ2: What is the impact of integrating Agile development and MDD on the software development process?
This research is conducted using the case study method [26]. We designed our study to compare the Agile MDD approach with three different approaches: MDD, Agile, and hand-coded approaches. In particular, the four applications were all implementing the same problem and were completed entirely independently using different development approaches by different developers:

1. A manually-coded version in C++, developed by a financial company.
2. An Agile MDD version, using UML-RSDS, developed by the second author.
3. An MDD approach, using ETL, developed by the fourth author.
4. An Agile approach, using Java, developed by the first author.

In order to answer the RQ1, the properties of quality, efficiency and maintainability of the case study versions were compared. For quality and maintainability, we use measures that are versions of the technical debt [16]. Technical debt is a metaphor referring to immature artifacts in the software development that negatively influence the software quality and maintainability in the long-run [23]. We have selected these technical measures because: 1) they have been used frequently in the literature [10,16]; 2) they are related to the quality of the software product that we intend to investigate, e.g., the complexity, coupling, and design flaws. We have used different measures and metrics to quantitatively compare all the applications by collecting the following data:

- **EAS:** Excessive application size. For MDD specifications a complexity measure can be defined, based on the total number of basic and composite expressions in the specification. For programs, LOC can be used. The threshold for a flaw to be present is a total complexity > 1000, or length > 500 LOC.
- **ENR:** Excessive number of rules ($nrules > 10$). This only applies to transformation languages such as ETL.
- **ENO:** Excessive number of helpers/operations ($nops > 10$)
- **EHS:** Excessive helper/operation size (complexity > 100 or length > 50 LOC). To give a more detailed comparison, we also consider maximum helper/operation size (MHS) within a program/specification.

- **ERS:** Excessive rule size (complexity > 100 or length greater than 50 LOC). Maximum rule size MRS is also considered.
- **EFO:** Excessive fan-out of a rule/operation (> 5 different rules/operations called from one rule/operation). Maximum fan out MFO is also considered.
- **CC:** Cyclomatic complexity (of rule logic or of procedural code) (> 10). Maximum CC MCC for any rule or code is considered.
- **CBR** Coupling between rules (number of rule/operation explicit or implicit calling relations > *nrules* + *nops*, or any cyclic dependencies exist in the rule/operation call graph).
- **DC:** Duplicate expressions/code (duplicate expressions or statements with token count > 10).

For efficiency, we measure the total execution time on a sample dataset input of 16 sectors and output of P(S = s) for all values of s $\leq$ 20 (Section 5). For size/complexity measures, we assess model-centric and code-centric approaches differently: in Agile MDD and MDD, the measures are applied at the specification level, while in Agile and the original manually-coded approaches measures are applied at the code level. **EAS**, **ENR** and **ERS** are omitted in both the Agile and original applications as they are not applicable. In order to answer the RQ2, we used qualitative measures through an opinion survey, in which the developers were asked to report in a free-form format the main benefits they perceived, along with the issues they faced in each approach.

4 Agile MDD Process Overview

In this section, we provide a brief overview of our Agile MDD approach. A more detailed description can be found in [2]. The development process starts with the *Initialisation* phase and finishes with the *Deployment* phase. 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*.

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 are identified and prioritised, and the product backlog is created.

Each iteration begins with an iteration planning activity to agree on the work to be accomplished in the upcoming iteration (the iteration backlog). The main process of an iteration involves three phases for each task in the iteration backlog:

- **Phase 1: Requirements Specification** The objective of this phase is to analyse and refine the functional and non-functional requirements from the

iteration backlog. Also, any existing components that can be reused in the current development or the potential for new components to be reused in the future should be identified where possible. Furthermore, the need for specific metamodels and transformations should be identified at this stage.

- **Phase 2: Development** The objective of this phase is to produce a complete and precise technical specification of the system and to produce (automatically or semi-automatically) an executable system that fulfils the functional and non-functional requirements. The specification should be reviewed and refactored continuously to ensure the best design of the system. Any changes to the requirements should be applied to the modelling level, that might imply changes to some products like models, metamodels, transformation, and updating new elements in the language.
- **Phase 3: Integration and Testing** In this phase, the developed parts of the system are integrated and tested.

Phase 4: Deployment
When the release has been fully implemented and has reached a stable version, it will then be deployed to the customer.

5 Case Study: Collateralized Debt Obligation (CDO)

The case study concerns the implementation of an application to calculate and evaluate the risk of financial investments known as *Collateralized Debt Obligation*s (CDOs) [9]. CDOs are composite investments whereby a portfolio of investments in different companies within different sectors is held as a single combined investment. These companies are usually organised into disjoint groups representing sectors (eg., insurance, entertainment, telecoms, etc), and there is the possibility of infection of defaults between different companies within each sector [7,9]. Risk analysis of a CDO involves calculating the probability $P(S = s)$ of the total credit loss s in the portfolio. In order to calculate the probability of the financial loss, the following formulas are used from *Hammarlid* [9]: Theorem 1.1, Theorem 3.1 and equations 1 and 2. These are listed below. The attribute k represents the index of one sector out of K total sectors, while the attribute n_k refers to the number of companies in sector k that are subject to risk. The attribute p_k represents the probability of a company defaulting in sector k while the attribute L_k represents the loss amount which is lost as a result of each default. The attribute q_k refers to the probability of infection of a default that may occur within sector k.

Theorem 1.1.

$$\mathrm{P}(\mathrm{N}_k = \mathrm{m}) = \binom{n_k}{m}\ (p_k^m(1-p_k)^{n_k-m}(1-q_k)^{m(n_k-m)} + \Sigma_{i=1}^{m-1}\binom{m}{i}(p_k^i(1-p_k)^{n_k-i}(1-(1-q_k)^i)^{m-i} \times (1-q_k)^{i(n_k-m)})$$

This equation gives the probability of m defaults in sector k. Conditioned on an outbreak in a sector, the distribution of the number of defaults is:

Equation (1)

$$P(N_k = m | N_k > 0) = P(N_k = m)/(1 - (1 - p_k)^{nk}) \quad (1)$$

and the probability of credit loss from sector k given an outbreak is:

Equation (2)

$$P(S_k = mL_k) = P(N_k = m | N_k > 0) \quad (2)$$

Theorem 3.1.
The overall probability of loss s from the CDO is given recursively by:

$$P(S = 0) = \exp(-\sum_{k=1}^{K} \mu_k) \quad (3)$$

and

$$P(S = s) = \frac{1}{s} \sum_{k=1}^{K} \sum_{m_k=1}^{[s/L_k]} \mu_k m_k L_k P(N_k = m_k | N_k > 0) \times P(S = s - m_k L_k) \quad (4)$$

The same mathematical specification was used for all the four versions of the case study.

5.1 CDO using Agile MDD approach

The CDO application has been implemented using UML-Rigorous System Design Support (UML-RSDS) [13]. UML-RSDS is based upon the use case, class diagram, and Object Constraint Language (OCL) notations of UML. These notations are used to write system specifications, and then a design expressed using UML activities is automatically generated from the specifications. Finally, executable code in many programming languages (Java, C++, and C#) can be automatically synthesised from the design. The customer of this application was a financial analyst working in a financial company. The customer's main requirement was to have a precise (but very computationally expensive) and an approximate version of the total credit loss P(S = s) to overcome the limitations of the current application used in the company (lacking efficiency and accuracy). The CDO was developed using the aforementioned Agile MDD (Section 4) by one developer who had 10 years experience of UML-RSDS and had no prior experience in financial applications.

The development process began by interviewing the customer to gather the requirements. Both the functional and non-functional requirements were identified and prioritised. Afterwards, the product backlog was created (see Table 1). Since, the financial domain was unfamiliar to the developer, a phase of background research was necessary to understand the problem and to clarify the required computations. The development was organised into four iterations, each of which resulted in the incremental development of the application. The user

Table 1: CDO Product Backlog for Agile MDD approach

ID	User story	Type	Priority
US1	As an investor, I want to compute the probability P(S=s) of a total loss amount of {s} from individual and infectious defaults within a CDO.	FR	1
US2	As an investor, I want to calculate the risk probability P(S>=s).	FR	2
US3	As an investor, I want to read the data from a CSV file containing the sectors and companies information.	FR	3
US4	As an investor, I want to receive the results in a file.	FR	4
US5	As an investor, I want to be able to receive the results in a practical time (less than 30 seconds for each {s} for a portfolio of 20 sectors and 100 companies).	NFR	1
US6	As an investor, I want the results to be accurate, within 5% of the theoretical exact results.	NFR	1
US7	As an investor, I want to handle the case of cross-sector companies and cross-sector infection.	FR	2

story US1 was further decomposed into *US1.1: calculate probability of no contagion* and *US1.2: calculate probability of contagion* and were developed during the first iteration. Both US5 and US6 (non-functional requirements) were considered while developing the corresponding functional requirements. Then US2 was performed in development iteration 2. During the development, a further external requirement was introduced by the customer to handle the case of cross-sector contagion (US7), and this was scheduled to be implemented in the third iteration. Finally, US3 and US4 – which both involve manual coding – were developed in the fourth iteration. The system specification (a class diagram and a use case) of the Agile MDD application is presented in Figure 1.

The specification of the user story US1 has the following postconditions (rules):

```
CDO::
  s : sectors => s.mu = 1 - ( ( 1 - s.p )->pow(s.n) )

CDO::
  ps0 = -sectors.mu.sum->exp()

CDO::
 Integer.subrange(0,20)->forAll( s | PS(s)->display() )
```

The first constraint initialises the *mu* attribute value for each sector. The second then initialises *ps0* using these values. The third constraint calculates and displays *PS(s)* for integer values s from 0 to 20. The operation *PS(s)* computes the Poisson approximation of the loss function, and is itself decomposed into computations of losses based on the possible combinations of failures in individual companies. *P(k,m)* is the probability of m defaults in sector k, *PCond(k,m)* is the conditional probability of m defaults in sector k, given that there is at least one default:

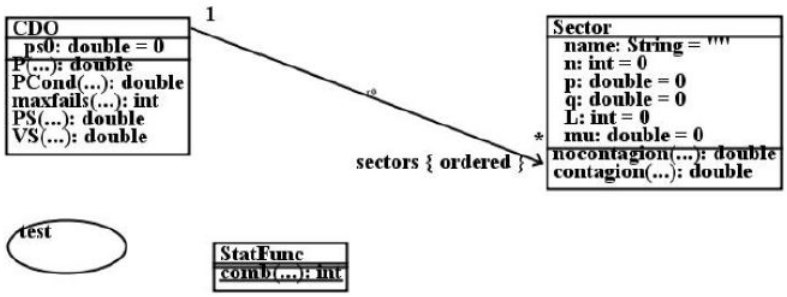

Fig. 1: The system specification for the CDO (Agile MDD application)

```
CDO::
query P(k : int, m : int) : double
pre: true
post:
  result = StatFunc.comb(sectors[k].n,m) *
 ( sectors[k].nocontagion(m) + Integer.Sum(1,m - 1,i,sectors[k].contagion(i,m)) )

CDO::
query PCond(k : int, m : int) : double
pre: true
post:
 ( m >= 1 =>
 result = P(k,m) / ( 1 - ( ( 1 - sectors[k].p )->pow(sectors[k].n) ) ) ) &
 ( m < 1 => result = 0 )
```

These correspond to Theorem 1.1 and Equation (1) above, respectively. For brevity, we only presented part of the system's specification, from which an executable implementation was automatically generated. The system was a full business success in the sense that it was delivered on time and it was more efficient than the one being used by the company. Subsequently, the final product has been successfully deployed to the customer.

5.2 CDO using MDD Application approach

The CDO case study was redeveloped using the same specification of [9], which have been presented above. The developer used EMF/ECORE [25] to specify the metamodels, and the transformation was implemented using the Epsilon Transformation Language (ETL) [12]. ETL is a hybrid model-to-model transformation language, that can transform many source to many target models. For this case study, single source and target models were used (the metamodel is presented in Figure 2). This solution was developed by one developer who had 4 years experience of ETL and had no experience in financial applications.

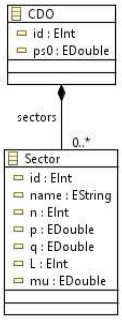

Fig. 2: The metamodel for the CDO (MDD application)

5.3 CDO using Agile approach

The CDO application was redeveloped using the Scrum process. The application was implemented in Java and the development process was organised in three iterations (each one-week long). Some Scrum techniques were used such as product backlog, sprint backlog, user stories, requirements prioritisation, sprint planning, sprint review, and frequent customer involvement. The developer had 5 years experience of Java programming and had no experience in financial applications.

5.4 The Original CDO Application

The CDO application was previously developed in C++ and used in a financial company. It was developed using a traditional code-centric approach (waterfall development model). The developer had over 20 years experience in C++ programming and was a financial analyst in that company. This original application is used as a basis for evaluation and comparison.

6 Results

We began our analysis by considering whether the developers in the three new case studies cover the full requirements (the original application was already developed and being used in practice). Both Agile MDD and Agile approaches delivered all the intended functionalities. However, the MDD development delivered an incomplete application, where some main functionalities such as calculating the total loss probability was missing. The main reason behind this failure was the lack of understanding of the requirements, due to the lack of close interaction with the customer. A month later, the developer succeeded to deliver a complete application after clarifying the requirements with the customer. To ensure that all the applications behave similarly, we executed each application on the same data test set of [9]. After the applications had been validated, we

compared the original CDO application with the three new versions developed using Agile MDD, MDD, and Agile, with regard to the quantitative measures of Section 3. For the Agile application, the measures have been computed using the source code analyser PMD, [4] whereas in the other applications, the measures were computed manually as there is a lack of tool support that can identify the technical debt in model-level artifacts (e.g. models, metamodels, and model transformation) [10].

Table 2 presents the execution time and the size in LOC of the four applications. The efficiency is investigated by measuring the execution time for uploading data input of 16 sectors and output of P(S $\leq$ 20), which involves computing $P(S = s)$ for each s from 0 to 20. The results show that Agile MDD application had the best efficiency (23 ms), while the original manual-code application had the slowest execution time with 231ms. The Agile and MDD applications had times of 39 and 123 ms, respectively. We also looked at $P(S \leq 50)$, however the two non-MDD applications did not terminate in a reasonable time for this test. The efficiency tests of Agile MDD, Agile and the manual-code applications were carried out on a standard Windows 8.1 with Intel i7 with 3.40 GHz processor, while the MDD test was carried out on Windows 7 PC using an Intel(R) Core(TM) i7 with 3.6 GHz processor due to technical issues. With regard to LOC, we measured the size by counting the number of lines of specifications in case of model-centric applications, while in code-centric applications (i.e, Agile and the original) we counted the number of lines of source code. Thus, the developer using Agile MDD produced only 94 LOC compared to 143, 196, and 129 for MDD, Agile, and the original applications, correspondingly.
Table 3 presents the values for each technical debt metric for the four applications. The metrics of *EAS*, *ENR*, and *MRS* were omitted in both Agile and the original applications as they are not applicable for code-centric estimation. One notable trend we can see is that Agile MDD has a clearly lower *EAS* or complexity (182) than the MDD application (693), whereas they have the same number of rules *ENR*. Regarding *ENO*, all the applications but Agile MDD exceeded the threshold of 10 operations. The other measure maximum *FO* (MFO) appeared to have the same value among all cases. While maximum *RS* (MRS) of the largest rule size (in LOC) did not occur in Agile and the original cases, the measure had a clear difference between Agile MDD (3) and MDD (18). Likewise, for maximum *HS* (MHS), the Agile MDD approach had the smallest value, whilst MDD approaches were better than non-MDD, and Agile approaches were better than non-Agile. When looking at the maximum *CC* (MCC), code-centric applications had the highest value for *CC* (6) while the lowest value occurred for Agile MDD with a value of 2. CBR is expressed as $CBR_1(CBR_2)$ where CBR_1 refers to the total number of calling dependencies between rules/operations, and CBR_2 is the number of rules/operations which occur in cycles of calling dependencies. The results show that Agile MDD had 11 dependencies between rules/operations and MDD had 13 dependencies between rules/operations (and each approach had two cyclic dependencies). On the other hand, dependencies between operations

[4] https://pmd.github.io/

Table 2: Execution time and LOC for each application

	Agile MDD	MDD	Agile	Original
Execution Time of P(S=s) for all s $\leq$ 20	23ms	123ms	39ms	231ms
Execution Time of P(S=s) for all s $\leq$ 50	93ms	531ms	$>$ 15min	$>$ 15min
Size(LOC)	94	143	196	129

were 7 and 11 in the Agile and the original applications, correspondingly, with only one cyclic dependency in each case. To identify design flaws, we consider if any bad smell occurs in the application, such as cyclic dependency, duplicated code, or any other violation for the technical debt thresholds. As seen in Table 3, Agile MDD had a lower number of design flaws, although there was not much difference in the flaws density (design flaws/LOC) in the four cases. The MDD applications had similar flaw density to the code-centric solutions, while the Agile applications had lower flaw density than non-Agile applications.

To complement our above quantitative results, the developers were asked to report in an opinion survey the main benefits and issues they perceived in each approach. Starting with Agile MDD approach, development time was reduced due to the direct feedback from the customer during development. In addition, the developer recognised development effort reduction and faster response to changes due to the small size and simplicity of the specification. With respect to the MDD approach, the developer reported as an advantage reduced development effort and high maintainability (related to system revision). On the other side, he faced difficulties in understanding the problem domain at the beginning. Finally, we asked the same question regarding Agile approach. The main benefit perceived by the developer is that frequent costumer involvement assured the application being built meets his needs. In addition, the iterative and incremental nature of the Agile development helped in organising the tasks. Although the application size is small, the developer reported a low maintainability as an issue during the development process. Opinion survey for the traditional code-centric approach was not feasible.

7 Discussion

In this section, we will structure our discussion regarding the impact of integrating Agile and MDD – in terms of the software quality and efficiency – in three ways: Agile vs Non-Agile, MDD vs non-MDD, and Agile MDD vs MDD approaches. In addition, other general insights about integrating Agile and MDD will be discussed.

In terms of efficiency, the application developed using Agile MDD was the fastest. However, the initial efficiency of the solution was too slow, as it took over 2 minutes to perform the calculation of P(S=s) for all values of s $\leq$ 20. To address this issue, the recursive operations and other operations with high usage

Table 3: Technical debt metrics for each application (flaws underlined)

	Agile MDD	MDD	Agile	Original
EAS (complexity)	182	693	NA	NA
ENR	4	4	NA	NA
ENO	9	11	12	11
MHS	9	18	21	34
MRS	3	18	NA	NA
MFO	3	3	3	3
MCC	2	3	6	6
CBR	11(2)	13(2)	7(1)	11(1)
DC	0	1	2	2
Design flaws	2	4	4	4
Flaws density	0.0213	0.0279	0.0204	0.031

were given the stereotype $\ll$ *cached* $\gg$ to avoid unnecessary recomputation. This stereotype means that the operations are implemented using the *memoisation* technique of [18] to store previously-computed results. Likewise, after applying the stereotype *@cached* to the operations in the ETL solution, the efficiency improved significantly from 101 minutes to 123ms. Although no caching has been used in either the Agile or original application, they showed better efficiency than the MDD application for $P(S \leq 20)$. The reason of the slowness of ETL is probably that ETL is an interpreted language and hence, it takes longer time to execute an application than UML-RSDS (Agile MDD case) which compiles specifications to a 3GL.

With regard to the product quality, the Agile MDD application had consistently better metrics. The simplest software metric is size (LOC). Smaller size of specification/implementation usually corresponds to a higher quality and reduced maintainability. The Agile MDD application was the smallest of the four versions. The results also show that model-centric approaches have more concise specification size (118.5 LOC in average) compared to non-MDD approaches (162.5 LOC in average). Working at a higher level of abstraction in model-centric approaches plays an important role in the amount of specification the developers need to write. Moreover, the Agile MDD application adopted "simple design" – an Agile practice that encourages minimising the complexity of designs – and this resulted in a smaller size of specification (in particular, the refactoring 'Move operation' was applied to reduce code complexity). The imperative style of ETL resulted instead in a substantially larger specification. Whilst UML-RSDS places operations within classes (as in Figure 1), the ETL operations are simply listed at one level of scoping in a transformation text. Thus, instead of a sector operation using the *self* object to access sector data, in the ETL version a global lookup for a sector identified by an index precedes any sector-specific functionality, resulting in larger and slower code. For example, the ETL version (written in EOL) of *PCond* is:

```
operation PCond(k : Integer, m : Integer) : Real
```

```
{ var secK =
    OUT!Sector.allInstances.selectOne(se|se.id == k);
  if (m >= 1)
  { return P(k,m)/(1-((1-secK.p).pow(secK.n))); }
  else { return 0; }
}
```

Regarding the EAS measure, it is interesting to note the significant differences in values between the Agile MDD and MDD applications, although UML-RSDS (Agile MDD case) and ETL (MDD case) have similar expression languages based on OCL. Although none of the applications exceeded the threshold of CC, Agile MDD exhibits the lowest maximum complexity in any operation or rule. Lower complexity should correspond to higher quality and make it easier to understand the specification. Some characteristics of the Agile philosophy support the assertion that programs developed using an Agile process have lower complexity than software developed using non-Agile processes [11]. We can see this is true for the Agile versus non-Agile approaches, although the MDD versus non-MDD distinction for maximum CC is more evident. The adoption of simple design and refactoring in the Agile MDD application resulted in lower complexity and a well-designed system with a lower number of design flaws. Usually, lower complexity tends to reduce the coupling of code or between objects: the higher the complexity the more coupled the code is. Although it has been proved that using *refactoring* contributes in improving code quality by reducing code complexity and coupling [19], the Agile MDD application had more coupling between rules/operations than the Agile application.

Table 4, Table 5, and Table 6 present the average of the values of the metrics for the pairs of approaches: Agile versus non-Agile approaches, MDD versus non-MDD approaches, and Agile MDD versus MDD approaches, respectively. Agile approaches therefore have better values than non-Agile approaches in 8 of the 9 measures while MDD approaches have better values than non-MDD approaches in 7 of the 9 measures. Finally, Agile MDD approach have better values than the MDD approach in all 9 measures.

Agile development relies on frequent interactions with the customer throughout the development process, to share information and provide feedback on what has been done and what to achieve in the next iteration of the development. This Agile characteristic was a significant advantage in our case. Since the financial case study involves highly-complex computations, collaboration with the customer was necessary to ensure that the developer understand the requirements precisely. As a result of lack of customer involvement in the MDD application, the developer did not come to know that there are some missing requirements until the application was delivered to the customer. On the other hand, frequent validation by the customer minimised the risk of building wrong functionalities, in both the Agile MDD and Agile applications. In the case of the original application, the developer was a financial analyst developing a system for his company. Another intrinsic value of Agile development is rapid response to change. For the Agile MDD and Agile applications, we found that the fact that the developer

was working in short iterations (in both approaches) and at a higher abstraction level and with concise specifications (in Agile MDD), also made it easier and faster to respond to changes, compared to non-Agile approaches.

The impact of integrating Agile and MDD on productivity is also an important factor to consider. One means to assess productivity is to measure the effort put into development, in person days or hours. In this study, the effort of development was not feasible to measure as the Agile MDD developer spent an initial stage of background research familiarising with the domain concepts and identifying the appropriate mathematical definitions to use. On the other hand, both the MDD and Agile developers were provided with the required background material and a precise problem description and hence spent less time in understanding the problem and started with the development process sooner. The MDD approach was faster than the Agile approach – however incomplete functionality was initially produced by the MDD approach, and the overall effort in the two approaches are similar once the work needed to complete the MDD version is taken into account.

Table 4: Agile versus non-Agile approaches

	Effic.	LOC	ENO	MHS	MCC	CBR	DC	flaws	flaws/LOC
Agile	31ms	145	10.5	15	4	9(1.5)	1	3	0.0207
Non-agile	177ms	136	11	26	4.5	12(1.5)	1.5	4	0.0294

Table 5: MDD versus non-MDD approaches

	Effic.	LOC	ENO	MHS	MCC	CBR	DC	flaws	flaws/LOC
MDD	73ms	118.5	10	13.5	2.5	12(2)	0.5	3	0.0253
Non-MDD	135ms	162.5	11.5	27.5	6	9(1)	2	4	0.0246

Table 6: Agile MDD versus MDD approaches for CDO

	Effic.	LOC	ENO	MHS	MCC	CBR	DC	flaws	flaws/LOC
Agile MDD	23ms	94	9	9	2	11(2)	0	2	0.0213
MDD	123ms	143	11	18	3	13(2)	1	4	0.0279

7.1 Outcome of research questions

Regarding the research question RQ1, we found that integrating Agile and MDD has a clear potential in developing small-scale but highly intensive computa-

tional applications. There are several improvements visible in the Agile MDD application, specifically in the quality and efficiency of the system (Table 3). Furthermore, our Agile MDD approach specifies a complete application in a single integrated model (class diagram plus use cases), which should facilitate maintainability and responding to change. For RQ2, we believe that building the system using iterative and incremental development helped the developer understand the domain and the requirements better and hence made it more likely that they will build the correct system. Moreover, continuous testing and frequent interaction with the customer resulted in an early discovery of defects or flaws and hence resulted in lower defects compared to the MDD application. MDD approaches have potential benefits of reusability of functionality compared to non-MDD approaches (eg., the specification of a Sector could be reused from this application in another financial context). By specification at a high level, it is also simple to add a mechanism such as caching by adding a stereotype to an operation. In manually-coded approaches the implementation of caching is non-trivial.

7.2 Threats to validity

The results of this study are particularly interesting as they resulted from a close-to-industry case study context. However, like most empirical studies, this study has some limitations. The first limitation related to the development team size. Agile methods emphasise on communication, people, and team collaboration. In this research, each of the four applications has been implemented by one developer, and thus it might have a potential impact on the quality of the application developed, and hence impact the evaluation of the case study. However, all developers participated in this study have a good experience working in Agile and/or MDD. Also, other studies in literature show that Agile development can be carried out successfully with solo-developer [20,21]. The second limitation relates to the inevitable differences in languages used for MDD versus non-MDD approaches as it was not possible to find developers who have approximate experience in the same language. Nevertheless, our recent study [14] found that UML-RSDS and ETL are rather similar in terms of fault density over large case studies and thus the impact of this differences on the results should be minimum. Another limitation concerns the generalisation of the results. Although the application used in this study is a real industrial application, implemented according to real customer requirements, we cannot generalise the results to different application types or sizes without more experiments. A possible threat stems from the method of measurement, which have been done partially with tools (i.e. the PMD analyser) and partially manually. Staron et al. [24] have shown that comparison of measures assessed by different tools is error prone. However, the small size of the application made it possible to calculate the values of the measures manually by at least two of the authors. The last threat to the validity is running the test case data on two different operating systems that might affect the evaluation of the efficiency of the applications. Nevertheless,

the specification of the two operating systems is very similar and has almost negligible impact on the results.

8 Conclusion

The aim of this paper was to provide a better understanding of how integrating Agile development processes and MDD could impact on the properties of the developed software. We have compared the experiences of four different independent development teams using different development methodologies. The results show some indicators that Agile MDD has improved the efficiency and the quality. We believe that this study is an early step in understanding the impact of integrating Agile development and MDD. Certainly, more research is required to further investigate its benefits or disadvantages. To this end, we intend to replicate this study using larger case studies with larger development teams using the same transformation and programming languages. Moreover, more measured properties such as productivity, time-to-market and comprehensibility of the developed code/specification should also be measured in future studies.

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. Hessa Alfraihi and Kevin Lano. A process for integrating agile software development and model-driven development. In *In 3rd Flexible MDE Workshop (FlexMDE) co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2017)*, CEUR Workshop Proceedings, pages 412–417, Austin, TX, USA, 2017. CEUR-WS.org.
3. Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.
4. Victor R Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, 1992.
5. Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.
6. 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.
7. M Davis, V Lo, et al. Infectious defaults. *Quantitative Finance*, 1(4):382–387, 2001.
8. Martin Fowler. The new methodology. *Wuhan University Journal of Natural Sciences*, 6(1):12–24, Mar 2001.
9. Ola Hammarlid et al. Aggregating sectors in the infectious defaults model. *Quantitative Finance*, 4(1):64–69, 2004.

10. Xiao He, Paris Avgeriou, Peng Liang, and Zengyang Li. Technical debt in mde: a case study on gmf/emf-based projects. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 162–172. ACM, 2016.
11. Daniël Knippers. Agile software development and maintainability. In *15th Twente Student Conf*, 2011.
12. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
13. Kevin Lano. *Agile model-based development using UML-RSDS*. Boca Raton: CRC Press, 2017.
14. Kevin Lano, Shekoufeh Kolahdouz Rahimi, Mohammadreza Sharbaf, and Hessa Alfraihi. Technical debt in model transformation specifications. In *Theory and Practice of Model Transformations*, 2018.
15. Anthony MacDonald, Danny Russell, and Brenton Atchison. Model-driven development within a legacy system: an industry experience report. In *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pages 14–22. IEEE, 2005.
16. Radu Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9–1, 2012.
17. Stephen J Mellor, Tony Clark, and Takao Futagami. Model-driven development: guest editors' introduction. *IEEE software*, 20(5):14–18, 2003.
18. Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19, 1968.
19. Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer, 2008.
20. Anna Nyström. Agile solo-defining and evaluating an agile software development process for a single software developer. 2011.
21. Tiago Pagotto, José Augusto Fabri, Alexandre Lerario, and José Antonio Gonçalves. Scrum solo: software process for individual development. In *Information Systems and Technologies (CISTI), 2016 11th Iberian Conference on*, pages 1–6. IEEE, 2016.
22. K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Agile Software Development. Prentice Hall, 2002.
23. Carolyn Seaman and Yuepu Guo. Measuring and monitoring technical debt. In *Advances in Computers*, volume 82, pages 25–46. Elsevier, 2011.
24. Miroslaw Staron, Darko Durisic, and Rakesh Rana. Improving measurement certainty by using calibration to find systematic measurement errorâĂŤa case of lines-of-code measure. In *Software Engineering: Challenges and Solutions*, pages 119–132. Springer, 2017.
25. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2 edition, 2008.
26. Robert K Yin. Case study research: Design and methods. *SAGE*, 2003(181):15, 2003.
27. Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE software*, 28(2):84–91, 2011.