Mod4J: A Qualitative Case Study of Model-Driven Software Development

Vincent Lussenburg[1], Tijs van der Storm[2,3], Jurgen Vinju[2,3], and Jos Warmer[1]

[1] Ordina
{vincent.lussenburg,jos.warmer}@ordina.nl
[2] Centrum Wiskunde & Informatica
{t.van.der.storm,jurgen.vinju}@cwi.nl
[3] Universiteit van Amsterdam

Abstract. Model-driven software development (MDSD) has been on the rise over the past few years and is becoming more and more mature. However, evaluation in real-life industrial context is still scarce.

In this paper, we present a case-study evaluating the applicability of a state-of-the-art MDSD tool, MOD4J, a suite of domain specific languages (DSLs) for developing administrative enterprise applications. MOD4J was used to partially rebuild an industrially representative application. This implementation was then compared to a base implementation based on elicited success criteria. Our evaluation leads to a number of recommendations to improve MOD4J.

We conclude that having extension points for hand-written code is a good feature for a model driven software development environment.

1 Introduction

Model Driven Software Development (MDSD) has gained in popularity the recent years. However, research that directly evaluates the application of MDSD in an industrial setting is still scarce. In this paper we present a qualitative case study to evaluate the use of the state-of-the-art model driven development tool, MOD4J[1]. We evaluate how well an application that is built using MOD4J fulfills all of its requirements.

This research project commenced just as MOD4J delivered a first, stable version suitable for production use at Ordina. In this version, there is modeling support for three out of four logical application layers and work is being done to support modeling of the presentation layer. We evaluate this early version because we are interested in improving MOD4J and the applications it generates [5]. The results may also influence the current development of the presentation layer which uses the currently existing functionality as foundation.

In the remainder of this section we first motivate and introduce MOD4J. Then we describe our method of requirement elicitation and product evaluation in Section 2. An in-depth qualitative analysis produces a lot of data and discussion. A full account can be found in [10]. Section 3 contains a summary with the most interesting observations. We discuss threats to validity and related work in Section 4 before we conclude in Section 5.

[1] http://www.mod4j.org/

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part II, LNCS 6395, pp. 346–360, 2010.

1.1 Motivating Model Driven Software Development at Ordina

This research has been conducted at J-Technologies, a division of Ordina employing personnel specialized in the Java programming language. The services offered by J-Technologies range from hiring out Java professionals to building and designing software in the in-house development infrastructure called Smart-Java. Smart-Java supports application development by offering the necessary infrastructure tools and services, such as version control, build servers, issue trackers, customized Integrated Development Environments (IDEs) and software artifact distribution. The majority of the applications built in the Smart-Java development infrastructure are web- or service oriented administrative business applications.

It is observed that Smart-Java applications were of very different overall quality and their development often suffers from suboptimal velocity, although they are technically very similar to each other. Even the skeletons of basic Create, Read, Update, Delete (CRUD) applications take a long time to be set up. Because Smart-Java is mainly focused on infrastructural services, it has proved to be hard to address these issues. Therefore, the decision was made to investigate the possibility of expanding the Smart-Java development infrastructure to a software product line [7].

As a first step, a multi-tier reference architecture [2] was designed based on the experiences of the leading architects over the last few years. A common, high quality reference architecture enables reuse among projects and addresses important choices regarding non-functional requirements valid for all developed applications. Other objectives and advantages were identified, such as a lower learning curve for developers, as they do not have to learn a new architecture for each project, and improved maintainability. Next, MOD4J—Model Driven Development for Java— was founded to design and implement a MDSD environment that can support the developers in writing applications within the context of this reference architecture.

1.2 Introducing MOD4J

MOD4J is an open source domain specific environment for developing administrative enterprise applications. It consists of a number of domain specific languages (DSLs) that are used to describe different aspects of administrative enterprise applications. Currently, there are four:

A Business DSL to model the domain of the application. This consists of specifying the classes, properties, associations and business rules of the domain.

A Data Contract DSL to define Data Transfer Objects (DTOs) on the domain objects.

A Service DSL for defining application boundaries. It encapsulates the application's business logic, controls transactions and coordinates responses.

A Presentation DSL to define user interface components of the application[2].

An important attribute of MOD4J is that it provides *extension points* at every layer for programmer to add Java code to specialize an otherwise fully generated application. The architecture of what MOD4J generates is depicted in Figure 1.

[2] This DSL is currently still under development.

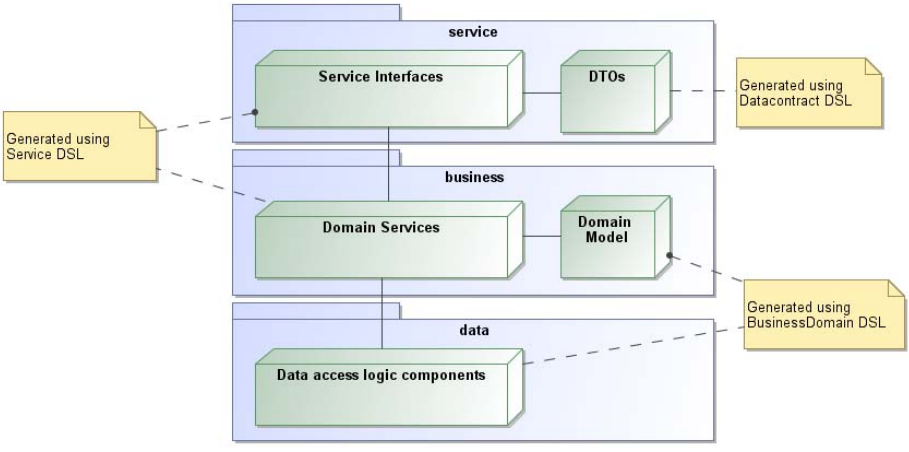


Fig. 1. Architecture of MOD4J-generated code

MOD4J is implemented using openArchitectureWare (oAW)[3], a language workbench [6] supporting activities ranging from the design of DSLs to code generation. The meta-models of the designed DSLs are used to generate rich text editors for use in the Eclipse IDE: they offer code completion, syntax highlighting and as-you-type validation. The XText module also allows validation rules to be specified for each DSL in both the (OCL-like) Checks language and plain Java. For the generation of the application code and configuration, MOD4J employs a Model-to-Text (M2T) approach [8,12] using the XPand component. The various layers are integrated by generating the configuration for the Inversion-of-Control (IoC) framework Spring[4].

2 Research Method

First, we discuss how we obtained the requirements, our evaluation criteria. Then we elaborate on the case we selected for evaluation. Finally we describe how we evaluate each criterion on the selected case.

2.1 Evaluation Criteria

Based on interviews and a dedicated workshop we have elicited the criteria that should make the use of MOD4J successful [10]. This has lead to the following three evaluation criteria:

1. Conformance to the reference architecture
2. Functional requirement satisfaction
3. Reduction of hand-written code

[3] Currently part of the Eclipse Modeling Project (`http://www.eclipse.org/modeling/`)
[4] `http://www.springsource.org/`

Conformance to the Reference Architecture. By definition, all products in the Smart-Java family must conform to the reference architecture [2]. This means MOD4J must support all common and variable features defined by it. This is an internal criterion, as MOD4J was designed to fulfill it. Nevertheless, the reference architecture was developed a priori so the question remains whether MOD4J-generated applications will meet its requirements.

Functional Requirement Satisfaction. We need to determine if MOD4J is suitable to be used to develop *any* product in the Smart-Java family. MOD4J raises the abstraction to a higher level and by implication limits expressiveness which might threaten the satisfaction of the more low-level functional requirements [12]. This is an external criterion: given any Smart-Java application will MOD4J be able to generate it?

Reduction of Hand-written Code. The goal of evaluating this criterion is to understand how much MOD4J will reduce the amount of developer effort. The amount of hand-written code is an easy-to-measure indicator of effort. If the amount of hand-written code would only be reduced marginally, this would invalidate investing in it. Note that a MOD4J application would contain code written in the four DSLs as well as Java code (in extension points). As an aside, we are also interested in the amount and quality of the generated code.

2.2 Case Selection

A representative application has to be selected and (re)built using MOD4J. To make sure we select a case that is independent of the design of MOD4J we have chosen an application developed a priori by Ordina ICT: JOBPORTAL (2006). An existing application also provides us with indisputable requirements to evaluate: the MOD4J version should simply do the same as the original JOBPORTAL.

JOBPORTAL was built using the Smart-Java development infrastructure. It is an application to support employees of a recruitment division in their work-flow. Types of activities include assigning an applicant to a recruiter, planning a meeting with the applicant, assigning the application to a reviewer for a review of the CV, maintaining the vacancies, etc. Also, people looking for work can search through the vacancies and send in an application. JOBPORTAL is implemented as a three-tier JEE application consisting of a data layer, a domain and service layer and two web applications.

JOBPORTAL is too big to re-implement completely (14.5k Non-Commented Source Statements (NCSS) and a total of 23 use cases). Instead, we have taken a sample of the total set of use cases this application provides. The goal of the use case sampling is to maximize the number of insights on working with MOD4J. The sampling technique we have used to accomplish this is called *snowball sampling.*. It was used to select use cases one-by-one allowing new insights during the implementation to dictate the choice for a next use case.

All this finally lead to the following selection:

UC02. Users should be able to search for a vacancy. The use case involves various custom search queries and works with a part of the domain model that is often used in the application. A portion of the domain model had to be modeled, which could be reused in subsequent use cases.

UC23. Recruiters should be able to maintain her own vacancies. This might reuse a part of the domain model that was constructed for UC02. As this use case creates, reads, updates and deletes data, it was expected to other new insights into modeling data modifications using MOD4J

UC11. Recruiters should be able to maintain reference data. This use case was chosen because we wanted to work with a new part of the domain model that still had some references to the existing domain model. The rationale for this was that it would yield information on how domain model partitions could be integrated.

2.3 Criteria Evaluation

Each criterion needs a method of evaluation, which we describe here.

Conformance to the Reference Architecture. In order to determine if the application built with MOD4J conforms to the reference architecture, we have first extracted the requirements from the reference architecture documentation [2]. In this document, the requirements have been laid down in an itemized, concise manner and therefore the conformance to them can be determined well in a source code walk-through session [11].

As an example, consider the following architectural requirement: *Domain objects must keep their internal state consistent*. Since MOD4J allows validation rules to be specified for attributes, such as optionality, maximum length, etc., such rules are fired each time an attribute is changed, keeping the internal state of the domain object consistent. This requirement is therefore considered fulfilled.

We introduce the following labels to assess to what extent a requirement is fulfilled:

Complete fulfillment. The architectural requirement is completely fulfilled by the code generated by MOD4J. Note that hand-written code or configuration can still violate the requirement.

Partial fulfillment. The rationale of the requirement is present but something is missing. This may occur when a single requirement defines several architectural rules of which some are fulfilled but others are not.

Violated. The architectural requirement is violated in the generated code. This label is given even when it is possible to correct or circumvent this violation by manually changing configuration code or implementing extension points.

Not at all. The architectural requirement is not addressed at all in the generated code, but developers may add hand-written code at a suitable extension point to fulfill this requirement. If this is not possible the requirement is considered *violated*.

Functional Requirement Satisfaction. Assuming the use cases we selected are representative, observations can be made regarding the fulfillment of the functional requirements relative to these use cases. We consider a use case to be implemented if the MOD4J-generated application has the same functionality as the original JOBPORTAL.

We have used existing correctness and completeness tests from the base application to find out which functional requirements have been implemented successfully. If an implementation is impeded or made difficult by MOD4J in any way, we collect a list of issues that cause this impediment. From this list we try to determine the root cause.

Reduction of Hand-written Code. Since we have not completely re-implemented JOBPORTAL simply comparing source lines is hard. We would need to extract the exact lines of source code that implement our use cases in the original application. This is difficult if not infeasible to do.

Instead we base our comparison on completeness and correctness tests. By running these tests in both the existing and the generated JOBPORTAL we are able to collect statistics about the source code that is executed for each use case.

Code coverage statistics are often used during unit testing to determine if all code is tested [16]. In our case, we use the coverage data to learn which byte code is executed for each individual correctness and completeness test, invoking both the MOD4J implementation and the original implementation. The resulting statistics are detailed enough to distinguish between the original implementation, the manual code and generated MOD4J code. These metrics are also used to determine the amount of hand-written code. The statistics are created by the free Eclipse plug-in EclEmma[5], a coverage tool that instruments byte code to show which code has been actually invoked by a certain execution.

3 Results

In this section, we will answer the question to what degree MOD4J meets the established criteria. The results have been obtained by completely implementing the three use cases using MOD4J and its extension points, and then applying the evaluation methods described above on the three selected criteria.

3.1 Conformance to the Reference Architecture

Research Data. The complete list of requirements consists of 72 requirements and can be found in the Appendix of [10]. In these tables we have provided the complete list of the requirements harvested from the architecture document [2]. The requirements are categorized according to *must*, *should* or *may* modalities. Here, we present three examples of requirements found in the architecture document:

- Domain objects *must* keep their internal state consistent.
- Data Service agents *should* encapsulate access to just one service.
- Domain objects *may* broadcast events about change in state.

Figure 2 provides a graphical overview of the results for all 72 requirements. Note that in this view, the requirements have *not* been weighted or prioritized and is therefore not suitable for drawing general quantitative conclusions. However, it does show that MOD4J does not fulfill all architectural requirements.

Evaluation. We argue that studying completely fulfilled architectural requirements and all *may* requirements will yield no interesting observations, unless they are violated. To filter out these requirements, we apply the filter in Table 1. The $\surd$ symbol indicates that requirements of the modality in a certain row are taken into account if it is fulfilled according to the predicate in each column. We summarized the determined causes of

[5] `http://www.eclemma.org/`

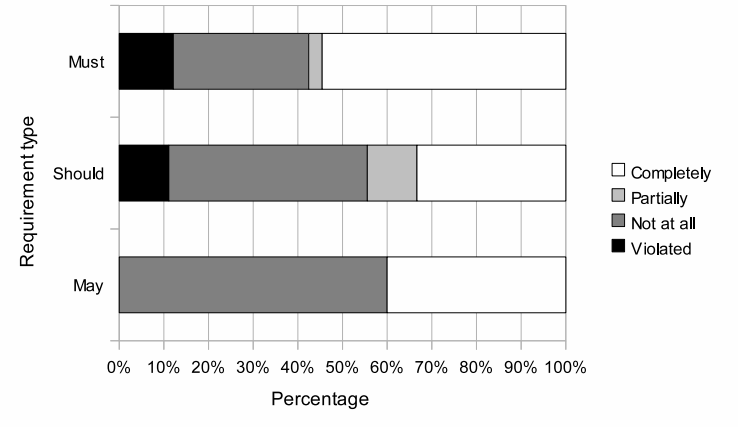


Fig. 2. Requirement fulfillment statistics

Table 1. Requirements filter

	Violated	Not at all	Partial	Complete
Must	√	√	√	–
Should	√	√	√	–
May	√	–	–	–

requirement violations in Table 2. We have seen that MOD4J follows the major parts of the reference architecture, but lacks modeling support for certain variable features [10]. The rows 2, 3, 5, 6, 7 from Table 2 describe the cause of this. These variable features can be implemented by hand using the aforementioned extension points, at a certain cost (see below where we evaluate hand-written code).

Cause #1 shows that the terms used in the four MOD4J DSLs do not map directly to the terms used in the architecture document. This may be expected in the business process layer, where most variability is to be expected between applications in the Smart-Java family.

Cause #4 we consider to be severe. Because the Business Domain DSL does not allow to define aggregate roots[6], MOD4J can not make the distinction between a high-level and normal domain object. Because this distinction can not be made, MOD4J treats every domain object as a high level object. This prevents modularization of the domain model, which may complicate maintenance. In MOD4J, this causes superfluous data access logic components which can not be removed by any means; they will always be generated even if they are never used. Other disadvantages include the inability to automatically delete all objects in the aggregate when the root is deleted. This has to be hand-coded in a Java extension.

[6] An aggregate root is a cluster of associated objects that is treated as a unit for the purpose of data changes [4].

Table 2. Summary of causes of architecture violations by MOD4J

#	Causes of architecture violations by MOD4J	Severity
1	The business processes follow a different nomenclature	–
2	The reference architecture is unclear on how to implement business work-flows.	–
3	Service agents can only invoked by business work flows and therefore are not supported as well. Also, implementation can vary greatly and should be occasionally used in the system, making it unsuitable for generation.	–
4	Business Domain DSL does not allow distinguishing between high-level and low-level domain objects.	+
5	Data service agents implementations can vary greatly and should be occasionally used in the system, making it unsuitable for generation.	–
6	MOD4J currently targets systems where the security concerns are addressed by the presentation layer.	–
7	Functional requirements for paging facilities are not clear.	–
8	Although a threat to conformance to the reference architecture, extension points are a necessary evil.	–

Table 3. Excerpt from [10]: functional issues with severity

#	Issue	Severity
1	Custom DAO implementation: can not disable generation of code and configuration	Major
2	Cannot not override boolean persistence configuration	Blocker
4	Binary data types are not supported	Blocker
14	MOD4J generates incorrect ORM mappings if a domain object has a many-to-many association with itself. Workaround in place.	Major
15	It is not possible to have a non-persistable domain object. Example: SearchResult. Persistence does not make sense here, yet mapping etc is generated.	Major
17	As the original service is the contract, the amount of service methods exposed in the MOD4J and original implementation should match up. In reality, the MOD4J service definition exposes more functionality.	Major
18	Cascading delete has to be hand-written for composite associations, introducing duplicate code (multiple domain services) or violating architectural requirements (calling other DAO's in a DAO).	Major

3.2 Functional Requirement Satisfaction

Research Data. We present the encountered functional issues in Table 3. They have been selected from the full record of issues [10] encountered during implementation if they satisfy the following two conditions:

- The issue describes a functional limitation or inability.
- Severity is *major* or *blocker*. Major means a hand-written extension was necessary. Blocker means that the issue prevents the satisfaction of a functional requirement completely.

Table 4. Functional requirements satisfaction summary

#	Observation cause	Severity
1	Persistence configuration is determined from the structural information laid down in the Business Domain DSL and application-wide properties. The offered flexibility does not suffice.	+
2	The Service DSL is unable to distinguish between domain service and local service.	−
3	The exact cause for the omission of binary data could not be determined from the research data.	+

Table 5. Byte code instructions executed (normalized)

(a) UC02 Select Vacancy

	original	MOD4J manual	MOD4J generated
data	346	115	93
business	0	141	31
domain	545	4	101
service	223	176	533
total	*1114*	*436*	*758*

(b) UC23 Maintain my vacancies

	original	MOD4J manual	MOD4J generated
data	331	156	102
business	0	5	23
domain	458	29	119
service	42	72	420
total	*831*	*262*	*664*

(c) UC11 Maintain reference data

	original	MOD4J manual	MOD4J generated
data	339	116	65
business	0	56	53
domain	373	0	156
service	70	92	627
total	*782*	*264*	*901*

(d) All three in one run

	original	MOD4J manual	MOD4J generated
data	787	387	176
business	0	202	107
domain	942	29	268
service	330	340	1154
total	*2059*	*958*	*1705*

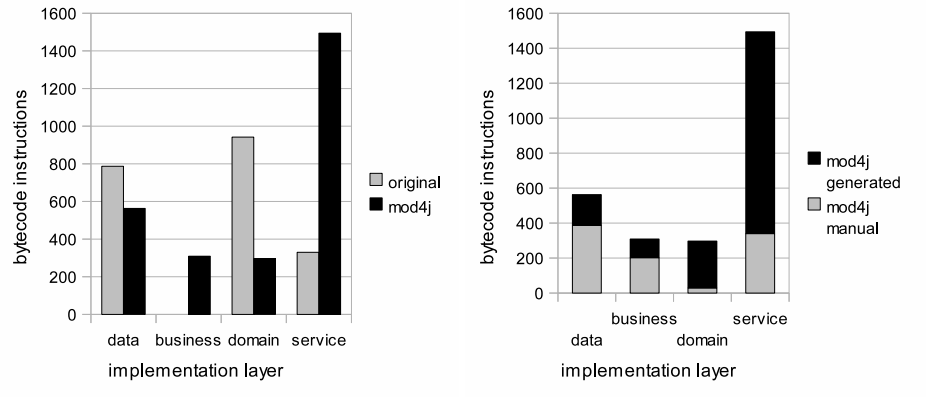
Evaluation. In Table 4 we have summarized and grouped the causes of the issues in Table 3. This Table is the result of manually comparing source code of the original application with the MOD4J-generated source code.

Note that the number of issues encountered while implementing the three selected use cases was quite minimal. Most of the original application could be implemented without any issue. Our choice to focus on even minor violations in this paper is motivated by our research perspective: we need to try and invalidate the claims of MDSD.

Still, we determined that issue causes #1 and #3 from Table 4 currently block the satisfaction of certain functional requirements in the JOBPORTAL case.

3.3 Reduction of Hand-Written Code

Research Data. The metric data of the covered code are provided in Tables 5a to 5d. This data has been normalized in order to do a fair comparison. For both the current and MOD4J implementation the start-up executes quite some instructions (constructors, initializers), which we removed. The rationale is that our implementation is only a partial clone of the original JOBPORTAL. Initialization code in the JOBPORTAL may easily be related to use cases that we did not select for this case study. Naturally, the unrefined metric data is also listed in [10].



(a) Original implementation vs MOD4J (b) Hand-written vs generated code

Fig. 3. Hand-written code charts

Evaluation. The following evaluation is structured along the design differences between the original JOBPORTAL and the MOD4J-generated JOBPORTAL. After that we evaluate the statistics regarding hand-written code versus generated code.

Design Differences. Figure 3a represents a graphical view of Table 5d:

- We see that MOD4J requires 2663 byte code instructions where the JOBPORTAL requires 2059 (table 5d). Based on these numbers MOD4J requires more byte code to execute the same business functionality. To determine what the cause of this is, we will zoom in on the differences for each layer.
- The original implementation requires more instructions in the data layer for the same functionality (figure 3a, data bar). The cause of the extra code in the data layer of the original implementation is that it has to adapt between behavior-less Transfer Objects used by the persistence framework and business objects used throughout the rest of the application.
- The original implementation has three times as much instructions in the domain layer (Figure 3a, domain bar). This is caused by the fact that MOD4J domain objects combine the original Business Objects and Transfer Objects resulting in a decline of code in the MOD4J domain layer compared to the original implementation.
- The original implementation has no executed instructions in the business layer at all (Figure 3a, business bar). This is caused by the fact that the original implementation has no business layer: the business process logic is coded in the service layer.
- The number of instructions required in the service layer is a factor four higher compared to the original implementation (Figure 3a, service bar). In MOD4J, domain object validation is done in the domain layer. In the original implementation, this is scattered throughout the domain model and service layer, resulting in a code shift from service layer to domain layer.

– MOD4J adds local services, DTOs and DTO translators in the service layer, resulting in a great increase of code in the service layer. The original implementation does not offer a specific course-grained interface and passes the business objects directly to the presentation layer. This means the presentation layer can directly execute business logic by invoking operations on these business objects, something that is not allowed in the reference architecture as all business logic has to be invoked through the service layer. MOD4J addresses a concern in the service layer that is not addressed by the original implementation.

Based on this analysis we conclude that the DTOs in service layer of MOD4J are the cause of the fact that MOD4J requires more code for the same functionality. Each DTO more or less duplicates the domain object and there can be, and often are, multiple DTOs for each domain object. For each DTO except custom DTOs there is a Translator that maps between the DTO and the domain object. The resulting amount of generated code is huge, as can also be seen in Figure 3b (service bar). The fact that MOD4J does not support modeling of aggregate roots required more CRUD methods to be generated, however, the resulting increase in code is negligible compared to the effect of DTO translators.

Hand-written Code vs Generated code. Figure 3b is another view on Table 5d, now focusing on the distribution of hand-written and generated code in the MOD4J implementation only. Based on this data, the developed application was analyzed.

1. MOD4J requires 958 byte code instructions from hand-written Java code, where the JOBPORTAL code is all hand-written, to a total of 2059 byte code instructions (Table 5d). Based on these numbers, we conclude that MOD4J has succeeded in decreasing the amount of hand-written code. Note that hand-written code compared to the original implementation is reduced by more than 50%. However, we also identified in the previous section that the service layer has no real equivalent in the original implementation. The actual gain is therefore even more because more then 50% of the hand-written code is located in the service layer. The actual reduction of hand-written code might therefore be as large as 75%. While the sample we have done is not by any means large enough for this conclusion to be statistically significant, it is hard data for a real case.
2. Of all the code in the analyzed sample, 64% is generated (Table 5d). This excludes start-up and initializing code as these have been subtracted during normalization. The unnormalized amounts would indicate that 71% of the total code is generated [10].
3. The data layer has a large amount (68%) of hand-written code (Figure 3b). Further analysis on the source code of the MOD4J implementation shows that this is caused by hand-written data access logic methods. We have spotted several opportunities of code that might also be generated, which are discussed in [10].
4. The domain layer is almost void (10%) of hand-written code (Figure 3b). This is caused by the fact that the JOBPORTAL functionality required little behavior and validation to be defined. Most functionality was implemented in the data layer, just like in the original implementation. An example is retrieving the Vacancies for a certain User: it is more efficient to determine this by executing a query on

Table 6. Observations on reduction of hand-written code

#	Observation cause	Priority
1	Custom methods require a lot of boiler-plate code, causing a large portion of the hand-written code. It may be an opportunity to support more methods to reduce the amount of hand-written code.	+
2	When using custom DTOs in a service method, the invocation to the domain service has to be hand-written due to the fact that the custom DTO can not be mapped onto a domain object. Since we argued that custom DTOs are not required, removing them and allowing simple types to be entered in the service method definition will result in less hand-written code.	+

the database then by traversing the entire object structure. Most validations in the domain model were quite simple (i.e., maximum length) and were automatically generated from the rules in the Business Domain DSL.

5. The business layer has a large amount (66%) of hand-written code (Figure 3b). Closer inspection of the implementation showed that some parts of this code pertained to boilerplate code that could have been generated Also, the large amount of hand-written code is to be expected, as the business logic should naturally be the most variable part of an application in the Smart-Java family.
6. The service layer has a considerate amount (23%) of hand-written code (Figure 3b). Contrary to expectation, a lot of boilerplate code to invoke the domain service had to be hand-coded.

We have summarized the severity of the identified observations in Table 6. We have seen that the amount of hand-written code, for the functionality that could be fully rebuilt, has decreased compared to the original, hand-written application. Although the Java extension points provided by MOD4J were used, they certainly did not result in a cost that would shadow the gain of code generation from abstract models. We did identify two opportunities, cause #1 and #2 in Table 6, to decrease the amount of hand-written code.

4 Discussion

An in-depth qualitative evaluation of a single software product such as JOBPORTAL does not necessarily lead to grand conclusions. Nevertheless, here we have extensively studied one application, generated for the most part by a state-of-the art MDSD tool. We have observed that a non-trivial application which existed a priori could be implemented using MOD4J, with a number of (minor) unforeseen issues.

Since we have not completely rebuilt the whole application, a threat-to-validity of the above observation is that we might have uncovered more vulnerabilities later. Our snowball sampling technique was designed to mitigate this effect. We feel confident that we have sampled a most difficult and relevant part of the application, but we must pay attention to this completeness issue nevertheless.

We have focused on three evaluation criteria that were elicited from a professional team at Ordina, using exploratory interviews and an in-depth workshop. We do not know whether our resulting focus on architecture, requirements and amount of hand-written source code is representative for other software development projects. Naturally, different criteria would lead to a different study of the quality of MOD4J.

We would like to observe that any software is subject to a changing environment and changing requirements [9]. We therefore believe that the existence of extension points for hand-written code, such as offered by MOD4J, is essential. It allows the programmer to adapt to changing or unforeseen circumstances without having to directly adapt the model driven development environment. In the JOBPORTAL case, extension points were used to implement features that were expected not to be generated, as well as features that might have been generated if MOD4J would provide such a feature.

Note that exactly these hand-written extensions are considered a bad thing from another viewpoint. We conclude that a model driven development environment should evolve with the applications that it generates. Frequent qualitative analyses such as performed in this paper, are necessary to update and refresh the model driven development environment such that the use of extension points will not start to outweigh the benefits of modeling and code generation.

4.1 Related Work

Smart-Microsoft. Warmer, the project leader of MOD4J, has designed a model driven software factory before: the Smart-Microsoft software factory. His experiences are described in [15]. In this paper, the chosen DSLs and architecture are explained in detail. Of course, Warmer's experiences have had a great impact on MOD4J and the DSLs are very similar to those in Smart-Microsoft. It would be interesting to compare the results from this study with the projects done in the Smart-Microsoft software factory. However, as the research assignment was primarily scoped on MOD4J we did not have the time to gather the project data as this was not readily available in the organization.

Generation of Web Applications. Visser [14] presents a case study in domain-specific language engineering. He designs and implements a number of DSLs which generate a web application for the full one hundred percent. Visser uses the SDF2 formalism to define a concrete syntax for the DSLs and term rewriting to generate code. While the case study in conducted in the same area MOD4J focuses on, the focus of the paper is quite different. Visser explains that, in the first place, his work is intended as a case study in the development of DSLs.

Although the approach of generation of web applications that Visser employs is comparable with the approach that MOD4J follows, Visser's conclusions do not directly overlap or contradict our own.

Changeability in Model Driven Web Development. Van Dijk [3] carries out an experiment to assess the changeability of model driven development of small to medium size web applications and compares it to the changeability of classically developed projects. He concludes that the changeability of web applications developed in model driven approach is competitive with classical approaches. The experiments are not carried out on real-life projects but rather on a small application developed by Van Dijk himself. Like

MOD4J, he uses the openArchitectureWare tooling to design the meta models of the DSLs and the templates used for code generation.

Because our study did not focus on changeability, Van Dijk's conclusions do neither contradict nor confirm our own conclusions.

Adoption of Model Driven Software Development. Related work on evaluating model driven engineering is relatively scarce. A large case-study to evaluate model driven engineering practices at Motorola is described in [1]. The study reports that an infrastructure division at Motorola achieved 65%–80% code generation by applying model driven engineering principles, leading to overall improvements in productivity and quality. These observations seem to be consistent with the results in this paper.

Staron [13] investigates the factors that influence the adoption of MDSD at two companies. One of the conclusions of this study is that modeling tools should be integrated in the software development process without completely redefining it. This is compatible with our observations as to how much MOD4J generated applications conform to the reference architecture of J-Technologies.

5 Conclusion

We have done an extensive evaluation of the suitability of MOD4J for building applications within a domain defined by a reference architecture.

By eliciting the criteria in a structured way, consulting both literature and experts, we have selected criteria for evaluating MOD4J. Next, we have set up a research method with a strong focus on data validity to evaluate these criteria. The study resulted in a number of issues and causes thereof which may be remedied in the design of MOD4J.

We conclude that MOD4J is suitable to be used to build applications that fall within the domain of the Ordina J-Technologies reference architecture. Since we have indications that up to 71% of the code can be generated, we think it probable that applications will be built with less effort than before.

We found that the use of hand-written code using the extension points of MOD4J was instrumental in implementing the fine details of some functional requirements. We propose that designers of model driven development environments introduce such a feature and at the same time try to prevent its usage by (incrementally) perfecting their modeling languages.

References

1. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context—Motorola case study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)
2. van Boxtel, P., Malotaux, E.J., Tjon-a-Hen, P.: Ordina Java Referentie Architectuur. Ordina J-Technologies, version 1.1 (2008) (in Dutch)
3. van Dijk, D.: Changeability in model-driven web development. Master's thesis, University of Amsterdam (2009), `http://dare.uva.nl/en/scriptie/313029`
4. Evans, E.: Domain-Driven Design: Tackling Complexity In the Heart of Software. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)

5. Fagan, M.: Design and code inspections to reduce errors in program development. In: Software Pioneers: Contributions to Software Engineering, pp. 575–607. Springer, New York (2002)
6. Fowler, M.: Language workbenches: The killer-app for domain specific languages? (May 2005), http://www.martinfowler.com/articles/languageWorkbench.html
7. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (August 2004)
8. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, Reading (2008)
9. Lehman, M.: On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software 1, 213–221 (1979)
10. Lussenburg, V.: Mod4J: A qualitative case study of industrially applied model-driven software development. Master's thesis, Universiteit van Amsterdam (2009), http://dare.uva.nl/en/scriptie/321845
11. Myers, G.J.: The Art of Software Testing, 2nd edn. Wiley, Chichester (June 2004)
12. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
13. Staron, M.: Adopting model driven software development in industry—a case study at two companies. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 57–72. Springer, Heidelberg (2006)
14. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)
15. Warmer, J.: A model driven software factory using domain specific languages. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 194–203. Springer, Heidelberg (2007)
16. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)