Mod4j:
A qualitative case study of industrially applied model-driven software development

V.Q.J. Lussenburg

v.q.j.lussenburg@student.uva.nl

October 27th, 2009

Master Software Engineering 2007-2009

Institution supervisor: dr. J.J. Vinju

Company supervisor: drs. J. Warmer

Availability: public domain

Institution:
University of Amsterdam
Faculty of Science
www.science.uva.nl

Company:
Ordina ICT
J-Technologies
www.j-technologies.nl

Abstract

Model-driven development has been on the rise over the past few years and is becoming more and more mature. This study offers insights into the applicability of the state-of-the-art model-driven software development (MDSD) tooling in real-life industrial context. This is done by means of an evaluation of the model-driven environment mod4j, a suite of DSLs focused on supporting the development of applications that fall within the domain of a reference architecture, featuring rich textual model editors integrated in the Eclipse IDE based on the latest MDSD best-practices and tooling.

First, the criteria that are critical to the success of mod4j are elicited in a structured way by consulting mod4j experts and literature. It is concluded that the model-driven tool should at least be able to satisfy the functional requirements of applications in the reference architecture domain. Also, basic non-functional requirements laid down in the reference architecture will have to be fulfilled. Finally, the amount of hand-written code is selected as criterion because it is a low-level criterion that enables high-level MDSD goals such as development velocity: if the amount of hand-written has not decreased compared to a completely hand-written application, it is unlikely the higher level goals can be satisfied.

The evaluation is carried out by means of a case study: an industrially representative application is carefully selected and partially rebuilt using mod4j. This implementation serves as case in this study. First, the artifact is thoroughly validated by means of an automated test suite and a walkthrough session with mod4j experts. It is analyzed to what extent the implementation meets the evaluation criteria and for each criterion, several issues are identified. For these issues, detailed recommendations are done to resolve them.

It is concluded that when these recommendations are followed, mod4j is suitable to be used to build applications that fall within the domain of the Ordina J-Technologies reference architecture. Because measurements showed that up to 71% of the code can be generated, it is considered probable that applications will be built in less time and with less effort.

Preface

First, I want to thank Jos: it has been a pleasure to have worked with you. It's amazing much knowledge on MDSD you have, I could have had no better mentor during this project. Also, I extend my gratitude to the rest of the mod4j team, Eric Jan and Johan, for investing their scarce time into contributing to this study.

Next, I want to thank Jurgen for all the help in designing and structuring my research over the past few months. Your experience in the research area and confidence in me have helped me a great deal.

Hans, thanks for all the work in the past two years coordinating the master programme and thanks a lot for standing in for Jurgen during his holiday and helping to get my research project on the rails.

I want to thank Katinka, my girlfriend. Your help, support and patience have really been invaluable to me. I am really lucky to have a girlfriend that is already a master of science and more than often our dinnertime conversations have been dominated by discussions on how to approach certain research problem.

Of course, I want to thank my brother, parents and friends for coping with me being a hermit for almost six months, listening to me and providing me with sorely needed distractions.

Last, I want to thank Ordina J-Technologies and Edwin in particular both for the support during my project and for giving me the opportunity to obtain my master's degree part-time.

Contents

1 Introduction . . . 1
2 On Mod4j . . . 3
3 Evaluation criteria . . . 7
4 Research method . . . 11
5 Research results . . . 16
6 Recommendations . . . 28
7 Threats to validity . . . 32
8 Related work . . . 32
9 Conclusion . . . 33
References . . . 33
A Architectural requirements . . . 36
B Functional implementation issues . . . 41
C Hand-written code statistics . . . 43

1 Introduction

1.1 Motivation

Model-Driven Software Development (MDSD) is gaining popularity as literature on how to employ it and tooling to support it is become more and more mature. However, not much research has been published that directly evaluates the application of MDSD in an industrial setting. We will contribute data by evaluating the applicability of a state-of-the-art Model-Driven environment by doing a case study in the context of a software factory.

1.2 Project context

This research project has been conducted at J-Technologies, a division of Ordina ICT 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 Developer Environments (IDE) 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 the developed applications are of very different overall quality and development 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 product line or software factory, in other words, moving from supporting only infrastructural concerns to being more actively be involved in individual project processes. As a first step, a multi tier reference architecture was designed based on the experiences of the leading architects over the last few years. A common 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 domain of this reference architecture. Together, these initiatives are believed to have the potential to take the Smart-Java development infrastructure to the next level of industrialized software development.

1.3 Project purpose

This research project commenced just as mod4j delivered a first, stable version suitable for production use. 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. The current version does fully support the modeling and generation of the data, business and service layers (see section 2.1).

As it is one of the major goals of mod4j to be used in the Smart-Java development infrastructure, J-Technologies is interested how suitable the current version of mod4j is. While it is hard to establish the suitability of a product that has not been used in real-life yet, the rationale for this that the earlier problems are discovered, the less effort it will be to correct them (Fagan, 2002). This can also influence the current development of the Presentation DSL which uses the currently existing functionality as foundation.

The objective of this research project is to reach conclusions and recommendations regarding the suitability of mod4j within the context of the Smart-Java development infrastructure. We will conduct an evaluation in order to accomplish this objective and will answer the follow central question in this thesis:

- What conclusions and recommendations can be reached regarding the suitability of mod4j for use in the Smart-Java software factory?
  - What criteria will be used to evaluate the suitability of mod4j?
  - What conclusions can be drawn regarding the suitability of mod4j in light of these criteria?

– What recommendations follow from the conclusions regarding the suitability of mod4j?

Scope Due to the fact there is no longitudinal data on the use of mod4j, the evolution aspect of MDSD is placed out-of-scope.

1.4 Theoretical framework

Based on the project context, we identify that *Product lines*, *Software factories* and *Model-Driven Software Development* form the theoretical framework used to approach this evaluation. In the next sections, we will discuss and ground these claims.

1.4.1 Product lines and software factories

This research is conducted in the context of the development infrastructure of J-Technologies of which the evolution into a software product line or software factory is envisioned. A software product line is defined by Greenfield and Short (2003); (2004) as

> A software product line is a production capacity for a family of software products.

A software factory is defined by Greenfield and Short (2003); (2004) as:

> A software factory is a software product line that configures extensible tools, processes and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling and configuring framework-based components

Greenfield and Short (2003); (2004) underline the role of MDSD in a software factory. Another important product asset of a software factory is a *product line architecture:*

> A product line architecture describes the common high-level design features of the products, distinguishing between common and variable design features (Greenfield et al., 2004)

We have mentioned the term *reference architecture*, which is defined by (Bass, Clements and Kazman (2003), chapter 2) as follows:

> A reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them.
>
> A reference model is a division of functionality together with data flow between the pieces.

The reference architecture is an intermediate stage in architecture design where the architectural pattern has been determined and some high level design decisions have been made, but it is not yet detailed enough to serve as an architecture to implement a product.

How does one interpret the J-Technologies reference architecture in light of these two definitions?

First, the reference architecture is indeed a reference architecture in terms of Bass, Clements and Kazman (2003): it is an intermediate stage in architecture design particular to the J-Technologies domain and can be reused to create architectures for new products. By setting a context and scoping the domain, it is much more detailed then the definition by Bass, Clements and Kazman (2003) suggests.

Second, the reference architecture can be used to serve as product line architecture: it defines rules and guidelines that must be followed for each product (common design features) but also prescribes how to implement features that may not be required for every product (variable design features).

We conclude that the reference architecture can also be seen as *product line architecture*.

1.4.2 Model-Driven Software Development

Model-Driven Software Development (MDSD) is a software methodology that aims at using models as primary artifact. It is synonymous with Model-Driven Engineering (MDE) and Model-Driven Development (MDD). A model is defined by Hailpern and Tarr (2006) as:

> A *model* M is an abstraction over some (part of a) software product (e.g., requirements specification, design, code, test, call-flow graph).

Bézivin (2005) states that everything is (or can be seen as) a model, just as every is an object in the object-oriented paradigm. In line with this, Kleppe (2008) coins the term *mogram* for a product written in any - graphical or textual - software language: a classical Java application is also a model.

Hailpern and Tarr (2006) state that the primary goal is to raise the level of abstraction at which developers operate and, in doing so, reduce both the amount of developer effort and the complexity of the software artifacts that the developers use.

MDSD most associated with, but not limited to, design of *domain-specific languages* (DSL) and using these to generate code (Stahl, Voelter and Czarnecki, 2006).

DSL A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain (van Deursen et al., 2000).

Through these techniques, development at a suitable level of abstraction is attempted to be accomplished.

1.5 Thesis structure

The remainder of this thesis is structured as follows:

- Section 2 provides in-depth information on mod4j.
- Section 3 elicits evaluation criteria and selects which will be used for this evaluation.
- Section 4 elaborates on how the evaluation will be done.
- Section 5 presents and analyzes the research results.
- Section 6 does recommendations regarding the identified severe issues.
- Section 7 reflects on the research method and points out threats to validity.
- Section 9 draws conclusions based on the research results and recommendations.

2 On Mod4j

In this section, we provide an in-depth outline of what mod4j is. The level of detail is required to understand the more technical issues we will discuss later in the study.

- We will first provide an outline of mod4j.
- Next, we will elaborate shortly on how mod4j is implemented
- Then, we will discuss the architecture of the applications generated by mod4j
- Also, we will discuss the domain and purpose of each of the three DSLs in mod4j
- Finally, we will explain how the generated code is integrated

2.1 Outline

In order to provide an outline of mod4j, we cite the website[1]:

> Mod4j (Modeling for Java) is an open source DSL-based environment for developing administrative enterprise applications. It uses a collection of DSLs to model different parts of the architecture, combined with manually written code. Currently Mod4j consists of four DSLs: the Business Domain DSL, Service DSL, Data Contract DSL and Presentation DSL[2]. The modeling environment is seamlessly integrated into the Eclipse IDE which gives the developers one environment where they can easily switch back- and forth between models and code. The different DSLs used in Mod4j can be used independently, but if they are used in collaboration they will be fully validated with each other. Apart from integration in the Eclipse IDE, the complete code generation process can be run automatically on a build server without the need for Eclipse.

[1] www.mod4j.org

[2] The Presentation DSL is still under development at this time.

2.2 Implementation

Mod4j is implemented using openArchitectureWare[3], a language workbench (Fowler, 2005) 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 (XText module) 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 (Kleppe, 2008; Stahl, Voelter and Czarnecki, 2006) using the XPand component: the DSL is parsed into an abstract syntax tree (AST) and is expanded into the concrete syntax of the target languages using XPand templates.

Finally, oAW is used to define a work-flow that will be executed when the project with the DSLs is built. This work-flow allows the execution of tasks to be customized and ordered. Examples of these tasks are the actual generation of code but also formatting code and XML.

The oAW suite is discussed by Kleppe (2008); Stahl, Voelter and Czarnecki (2006); Breslav (2008) more extensively.

2.3 Architecture

The DSLs designed by mod4j are horizontal which means they apply to a technical domain instead of a business domain (Kleppe, 2008; Czarnecki and Eisenecker, 2000; Stahl, Voelter and Czarnecki, 2006). The rationale of this design decision is that applications built by J-Technologies do not have a common business domain but do have a common technical domain. As is visible in figure 1, each DSL is responsible for generating code of specific layers of the application as well as the configuration to integrate these layers. This way, a brand new project is quickly runnable without requiring any manual code or configuration.

Each of the three DSLs is designed to be highly cohesive and loosely coupled. The high cohesion results in a DSL to be focused on a single aspect of the technical domain. The low coupling prevents the DSLs from depending directly on the implementation of other DSLs, instead, an interface is exchanged. This allows referencing elements from one DSL in another DSL, enabling functionality like code completion and validation: this is what was meant by the statement that the models are fully validated with each other. Both separation of concerns and interface sharing are DSL design patterns described by Spinellis (2001).

2.4 BusinessDomain DSL

The BusinessDomain DSL allows the developer to model the domain of the application. This consists of specifying the classes, properties, associations and business rules of the domain. An example:

```
class Car [
    string brand;
    string model maxlength 20;

    rules [
        unique [brand model];
        TireValidationRule;
    ]
]

class Tires [
    integer size;
]

association
    Car car one <-> many Tire tires;
```

The business domain model generates code in the data and domain implementation layers. An example is provided in figure 2, the operations in the extension points (drive, inflate, deflate) are provided as examples of hand-written functionality. Note that the *data access logic component* for the Tire object has been omitted for brevity. The technique used to separate the generated from the hand-written code is called the Generation Gap pattern (Vlissides, 1996).

We see that for all modeled classes, a domain class and an extension point is generated, where manual code can be placed to extend the domain object, for example, behavioral business code. In the data layer, the object-relational mapping[4] configuration and *data access logic*

[3] http://www.eclipse.org/workinggroups/oaw/

[4] Object-relational mapping technique for adapting data between relational databases and object-oriented programming languages.

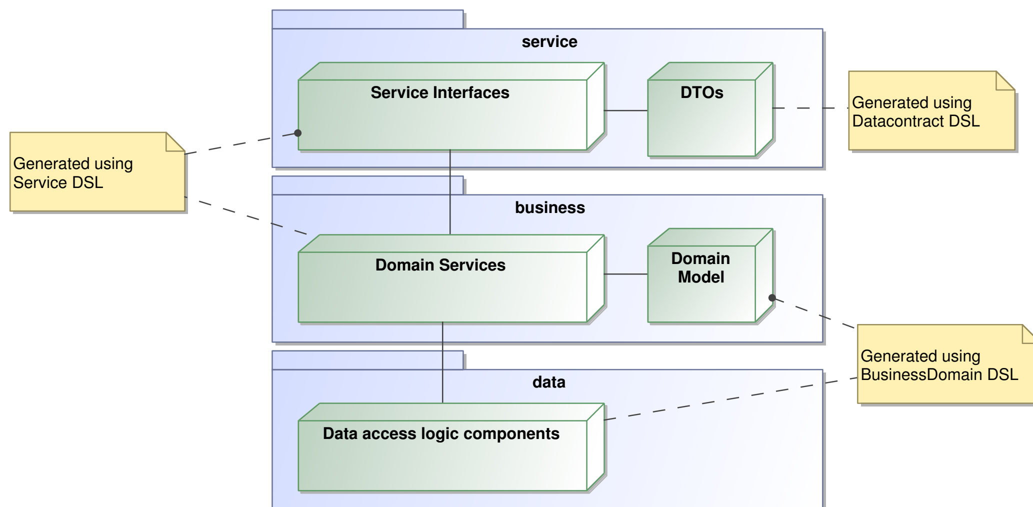


Fig. 1: Logical view mod4j architecture

component are generated for this domain class. Finally, a stub validation rule is generated in the business layer for each business rule that is specified in the model: this rule must be implemented by that. All extension points are generated by mod4j one first time if they do not yet exist: manual code in these files is never overwritten.

2.5 Service DSL

We will first discuss the definition and responsibilities the service layer. Fowler (2002) defines the role of service layer as follows:

> A service layer defines an application's boundary (Cockburn, 1996) and encapsulates the application's business logic, controls transactions and coordinates responses

He also underlines that the service might have to support remoting but recommends not doing this if it can be avoided as per the First Law of Distributed Object Design: Don't Distribute Your Objects.

In his Domain-Driven Design book Evans (2003) defines a domain service as follows:

> A domain service is a standalone, stateless interface that houses operations that are not a natural responsibility of an *Entity* or *ValueObject*. The domain service name as well as the operations should be part of the *Ubiquitous Language*.

As the name suggests, the domain service is part of the logical business layer. Any concerns regarding remoting and messaging are separated from the domain service: are classical example of separation of concerns (Dijkstra, 1982).

The mod4j architecture follows this: the domain service is part of the logical business layer and addresses the business logic concerns. The mapping between the external and the internal representation is addressed by the Datacontract DSL (see the next section). Mod4j currently exposes the functionality using a local service implementation: as we will use the term 'local service' often in the study, we will define it here explicitly:

> A local service is a service that uses a local (ie. non-remote) protocol to expose the business functionality in the domain service.

In summary, the domain service addresses business concerns, where the local service addresses technical format and transport concerns.

The Service DSL models the local service and not the domain service and can only reference DTOs. It may be contrary to expectation that the domain service is also generated: mod4j supports predefined common methods such as create, read, update and delete and

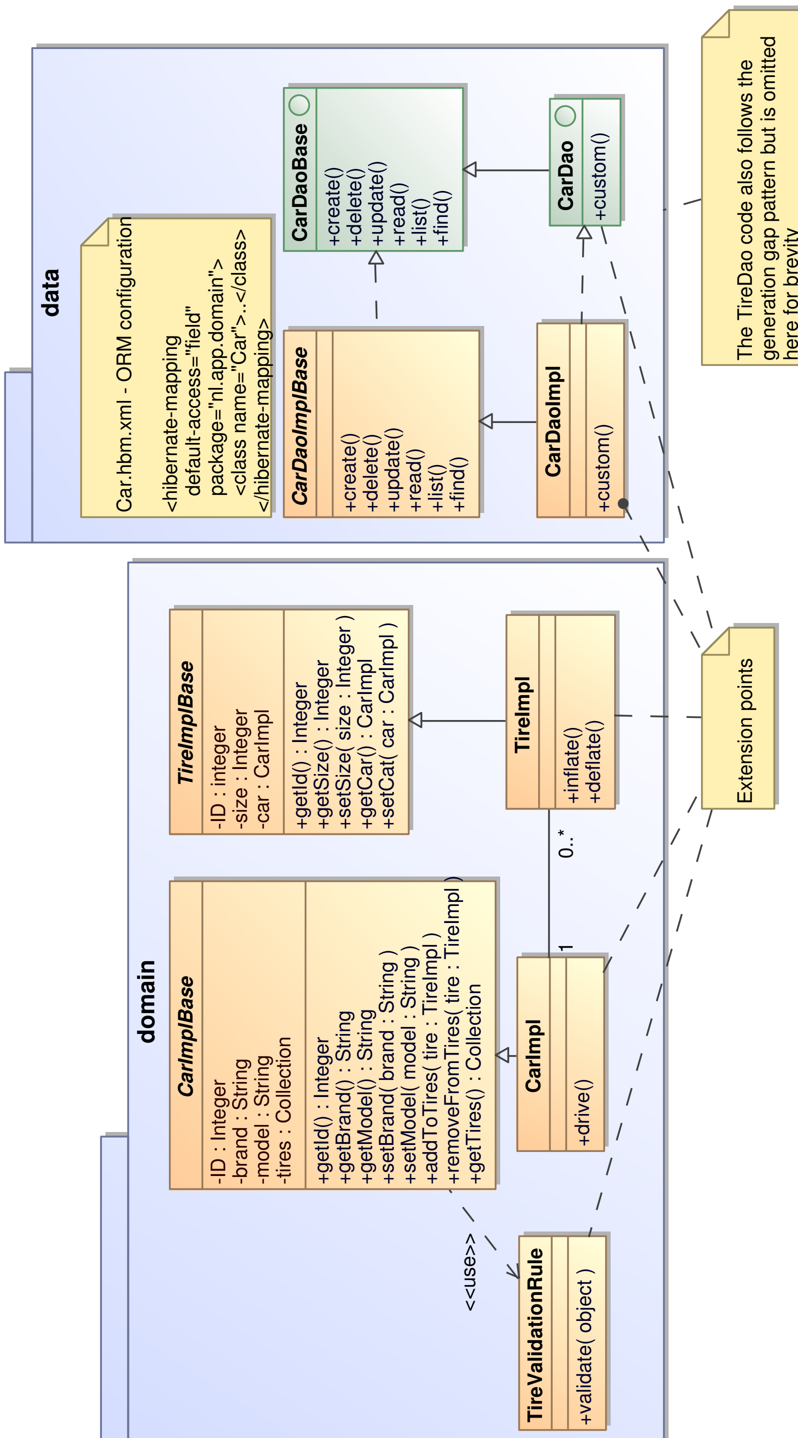


Fig. 2: Code generated from the BusinessDomain DSL (UML2 class diagram)

for these methods, the full implementation is generated from service layer to data layer, this requires that the domain service is also generated. This coupling between local and business service is also addressed in this study.

An example of the concrete syntax is provided in the program listing below, figure 3 depicts the code mod4j generates from this model.

```
"Read operation for CarWithTiresDto."
read readCarWithTiresDto for
    CarWithTiresDto;

"Lists all CarDto objects."
listall getCarList for CarDto;

"Get number of tires on the car"
method getNumberOfTires in
   [ CarDto carDto; ]
   out NumberOfTiresDto;
```

2.6 Datacontract DSL

The Datacontract DSL allows *Data Transfer Objects (DTOs)* to be defined based on the domain objects. The definition of DTO we will employ in this study:

> A *Data Transfer Object* is an object that carries data between processes in order to reduce the number of method calls (Fowler, 2002).

An elaborate explanation of the DTO pattern can be found in chapter 15, Fowler (2002). In short, the DTO is used to provide a course-grained, serializable, behaviorless object tailored towards to need of the client.

Mod4j allows to define DTOs that directly represent a domain object. It can be configured which properties and associations are to be included in the DTO. The DTO Translators that map between domain objects and DTOs are also generated. There is also the possibility to define a custom DTO that does not represent a domain object directly (see *NumberOfTiresDto* below). We will evaluate the role of the custom DTO in-depth in this study.

As an example of a datacontract model, we prevent the following program listing and figure 3 depicts what mod4j generates from this model.

```
"CarDto, only exposes the model property"
class CarDto represents Car [
   model;
]

"Dto exposing all properties and
      no associations."
class TireDto represents Tire

"A list of CarDto objects."
list TireDtoList contains TireDto

"CarDto, with tires association."
class CarWithTiresDto represents Car [
         model;
         references [
                  tires as TireDtoList;
     ]
]

"Custom DTO used to return the number of
     tires."
custom NumberOfTiresDto [
         integer numberOfTires;
]
```

2.7 Integration of layers

The various layers are integrated by generating the configuration for the Inversion-of-Control (IoC) framework Spring[5]. IoC injects dependencies into components instead of letting the components resolve their dependencies themselves. IoC and Spring in particular is very popular in JEE community. For further reading, see (Fowler, 2004).

3 Evaluation criteria

In this section we will answer the question what criteria will be used to evaluate the suitability of mod4j.

First, we have studied literature on all sections of the theoretical framework (section 1.4). Also, we have undertaken both an interview with the project lead, Warmer, and a workshop with the entire mod4j team in order to elucidate all the goals of mod4j and all problems it is envisioned to solve. We will elaborate on these

[5] http://www.springsource.org/

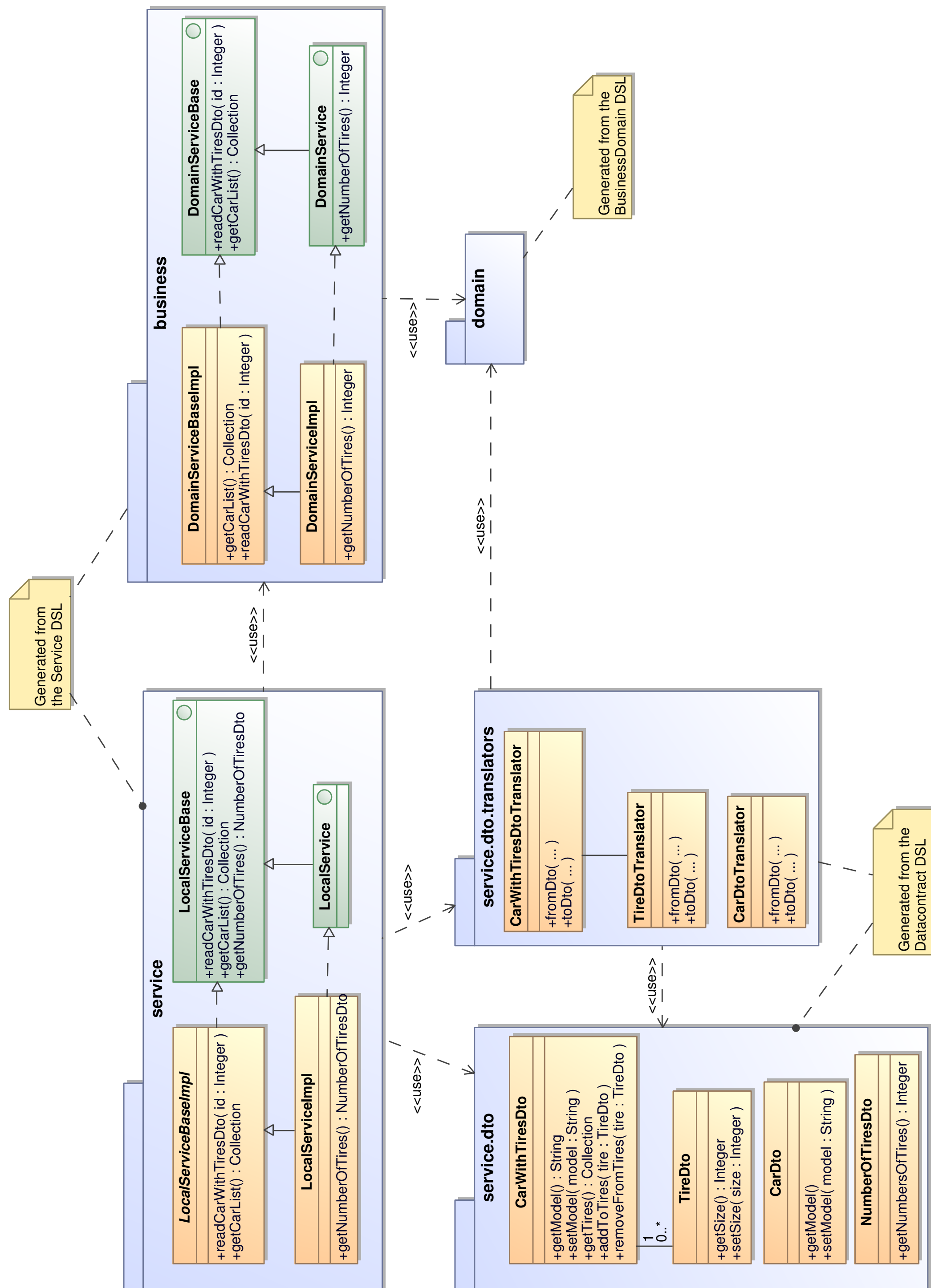


Fig. 3: Code generated from the Datacontract and Service DSLs (UML2 class diagram)

sessions first, after which we will present and ground the selected criteria and in the following order:

- Conformance to the reference architecture
- Functional requirement satisfaction
- Reduction of hand-written code

3.1 Criteria elicitation

3.1.1 Interview project lead

An interview with the project lead, Warmer, was setup early in the project to serve as a source of both contextual and in-depth information for the researcher and to elicit criteria for the evaluation. As an expert on both MDSD and mod4j Warmer was expected to have valuable input for an initial focus. The interview was structured around the following central topics:

- Mod4j: background, history, design, implementation
- Influences from previous MDSD experiences
- Reflection and vision on mod4j
- Suitable applications to build with mod4j

Especially during the first interview question Warmer elaborated on the problems that mod4j is envisioned to solve. Also, previous experiences of Warmer regarding a horizontal DSL suite to build administrative business applications (Warmer, 2007) yielded important data about how developers worked with the DSLs: what they liked but also what problems they encountered. These strong points and problems form a valuable viewpoint on mod4j.

The following items have been distilled from the interview transcript:

Conformance to reference architecture A corner stone of mod4j is that, out of the box, it generates an application that conforms to the architecture as laid down by the J-Technologies architects. In classical software development it often occurs that the reference architecture is violated, for example, because an architecture layer is illegally skipped. During the development of mod4j, the reference architecture has always been leading. This means that the architectural concerns are separated from mod4j: if the functional and non-functional requirements on applications can be implemented using the reference architecture, they can be implemented with mod4j too.

A result of the conformance of the architecture is a consistency in the codebase: One reason that project results are not predictable is that the choices made are different for each project.

Reduction of complexity The primary goal of MDSD is to raise the level of abstraction at which developers operate and, in doing so, reducing both the amount of developer effort and the complexity of the software artifacts that the developers use (Hailpern and Tarr, 2006). This goal applies to mod4j as well: the mod4j environment allows the developer to develop at a higher level of abstraction and be more productive.

Educational code The code generated by mod4j communicates the reference architecture and serves as a tool to teach new developers about the architecture. In Warmer's experiences, developers gain a lot of insights by creating a new toy application, working with the DSLs and watching the code that is generated. The code that is generated by mod4j is well-readable and rich in documentation to optimize this learning effect.

Textual editors In Warmer's previous experiences with model-driven development of administrative business applications, he has noticed that graphical DSLs do not match the developer's style of work: they are more accustomed to and comfortable with textual DSLs. It is expected that textual editors increase both the productivity and chance of adoption.

Suitable for the entire reference architecture Mod4j is suitable for all applications that fall under the domain of the reference architecture. For some applications the advantage of using mod4j will be greater then others, but all applications will benefit from it.

Velocity One of the most important goals of mod4j is to increase the average developer's velocity. This envisioned increase velocity should be accomplished through

mod4j's various other characteristics such as consistency and operating at a higher level of abstraction. While this is aggregate of already mentioned criteria, is it a very important metric to determine the success of mod4j.

3.1.2 Workshop

Introduction After the interview we conducted a workshop that was attended by the entire mod4j development team. The workshop was structured as follows:

- Research project outline
- Problems solved by mod4j
- Selected case outline
- Evaluation criteria elicitation

Problems solved by mod4j In order to get a good overview of all problems that mod4j tries to address, we asked the participants to write down the most important problems that mod4j will solve on a post-it note and them stick them all together on a flip-over. Next, we grouped similar problems and discussed all problems in order to provide context for them. This part of the workshop gave valuable insights into mod4j and the problems that are envisioned to be resolved.

- Consistency and quality: It is envisioned that mod4j will cause the overall quality of the source code to increase and let this increase be consistent over the entire codebase.
- Decrease required detailed knowledge: when working with mod4j, developers will need less detailed knowledge of the underlying frameworks and tools. This makes developers more productive.
- Velocity: mod4j means quicker development. This means the feedback loop is short and the project can be agile.
- Bring the IT and the business closer together: By increasing the velocity it can more quickly be verified if the customer is well understood by demonstrating a crude prototype. This prototype can both be made very quickly from scratch by mod4j and easily be expanded to a production-quality implementation.
- Maintainability: because the developer work at a higher level of abstract, the maintainability is envisioned to be higher.
- Architectural guarantee: Mod4j makes sure that the generated application matches the architecture. This means misinterpretations will no longer cause the architecture to be violated.

Evaluation criteria elicitation After the selected case had been outlined, the participants were asked once again to write down their thoughts but now on criteria that would be particularly interesting for the case to be studied. Again, the thoughts were grouped and discussed in depth.

- Flexibility: is mod4j flexible enough to work with existing designs? Does it integrate with an existing database schema, service model, et cetera?
- Best-practices: can the best practices of software development be followed? Or is another working style required?
- Maintainability (again).
- Broad applicability: can all applications within the reference architecture domain be implemented with mod4j?
- No impact on existing production environment: the created application should be just another JEE application.
- Easily usable (low learning curve).

3.2 Criteria

By weighing the input of the interview, the workshop and the literature on the various subjects of the theoretical framework (section 1.4) we have selected three criteria we will use to conduct the evaluation. These are:

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

We will discuss these criteria and explain why they have been selected for this evaluation.

3.2.1 Conformance to the reference architecture

The goal of evaluating this criterion is to determine if mod4j is suitable to be used to develop all products in the Smart-Java product family. This was mentioned both the in the interview with Warmer and in the workshop with the mod4j development team. Literature also underlines the importance of a *product line architecture* for model-driven software development (Greenfield et al., 2004; Stahl, Voelter and Czarnecki, 2006).

As we identified in section 1.4, Smart-Java is a development infrastructure aspiring to grow into a software factory. In this context, the reference architecture (described in Boxtel, Malotaux and Tjon-a Hen, 2008) can be seen as the *product line architecture* used for all products in the product family (Greenfield et al., 2004). As it is one of major goals of mod4j is to be used in a software factory, it must be able to support the development of all these products. This means mod4j must support all common and variable features defined in the reference architecture.

The initial goal of mod4j has been to deliver added value by supporting the most common tasks and not addressing all variable features. This is not an uncommon approach in code generation: for example Stahl, Voelter and Czarnecki (2006); Czarnecki and Eisenecker (2000) recommend not insisting on generating 100% of the code and rather focusing on where code generation delivers its most added value and making sure this generated code can be well integrated with the hand-written code. Even if a variable feature cannot be modeled in mod4j, it must be possible to hand-code this feature without having to change generated code or violate the reference architecture.

Based on these facts, we consider conformance to the reference architecture to be critical success factor for adoption in the Smart-Java development infrastructure and will therefore employ it as evaluation criterion.

3.2.2 Real-life functional requirement satisfaction

The goal of evaluating this criterion is the same as the previous: to determine if mod4j is suitable to be used to develop all products in the Smart-Java product family. However, this criterion approaches suitability from a different viewpoint. This has been mentioned neither in the interview nor in the workshop, because it is very implicit: it is evident that mod4j has to be able to statisfy real-life functional requirements for it to be practically usable for J-Technologies. Mod4j raises the abstraction to a higher level but also limits expressiveness which can threaten the satisfaction of the more low-level functional requirements (section 8.1, Stahl, Voelter and Czarnecki, 2006).

In the previous section we already identified that it is neither expected nor required to be able to model every facet of the the required functionality as long as it is possible to place hand-written code at suitable places within the architecture (Stahl, Voelter and Czarnecki, 2006). Even when this is made plausible, skepticists will insist on having success stories of the implementation of real-life requirements. By evaluating this criterion set the first step towards achieving this.

3.2.3 Reduction of hand-written code

The goal of evaluating this criterion is to determine if it is plausible that mod4j can reduce the amount of developer effort. This criterion was indirectly mentioned in both the interview and the workshop. The amount of hand-written code is an easy collectable metric and if the amount of hand-written code is not reduced, it is also unlikely that velocity of development is increased.

Also, by analyzing the hand-written code opportunities can be spotted to decrease the amount of hand-written code by adding or improving DSLs and/or generators.

4 Research method

In this section, we will determine which research method we will employ to evaluate mod4j. We have decided to do a case study because:

- The research is focused on a scoped domain.
- The nature of our study is more in-depth than broad.
- We have to do much work to rebuild a case: there is overhead in selecting multiple cases (finding suitable cases, familiarizing ourselves with the application, et cetera).
- We will collect qualitative rather than quantitative data.

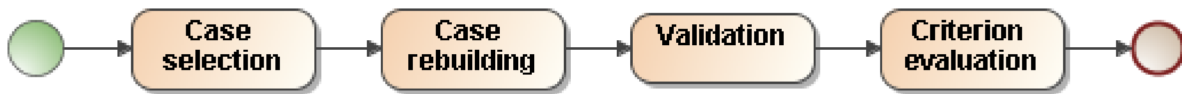


Fig. 4: Research method work-flow (Business Process Diagram)

The four steps of the research method are displayed in figure 4. We will first discuss them briefly.

1. Case selection: first, we will determine which case we will study.

2. Case rebuilding: here we will decide how to rebuild the case, including the selection of parts to rebuild.

3. Validation: here, we will address the concern that the created implementation is neither tested nor accepted by any customer and, as a result, might contain undiscovered bugs and missing or misinterpreted functions.

4. Criterion evaluation: here, we will determine how the data will be analyzed in order to reach conclusions and recommendations for each criterion.

4.1 Case selection

Mod4j is has not been industrially applied yet and therefore no real-life project data is yet available. The first step in our research method (figure 4) is to select a real-life, representative application which we can rebuild ourselves to acquire research data. We will select an application to rebuild with mod4j by first determining the criteria of a representative application and then selecting a suitable case based on these criteria.

4.1.1 Criteria

As defined in the previous section, the application needs to be representative within the context of J-Technologies. As this a broad definition, we reverse it and ask the question: what application would *not* be suitable for this study?

Conforming to reference architecture Any application that has little to no similarities with the reference architecture, either caused by the domain of the application (e.g. real-time, embedded) or design choices (e.g. no separated domain model or data layer), will result in work into adapting the design to match the reference architecture. This work does not directly contribute to the project goals and will introduce an uncertain factor.

No toy application A toy application is created to prove a concept or set an example but does not satisfy any real user requirements. Because no real user requirements are satisfied, these applications have a low complexity and contain undiscovered bugs. Although a good design aims at building "the simplest thing that could possibly work" (Beck and Andres, 2004), mod4j must be able to cope with situations where the simplest thing is not dead simple. We conclude that any observation done rebuilding a toy application is unlikely to be representative and will threaten the validity of the study.

In production use (mature) During production use, bugs are found and new insights are obtained regarding application features (Bennett and Rajlich, 2001). Hence, an application that has undergone some maintenance is more suitable than an application that is very new.

Not small The selected application should not be small as interesting complexity often emerges with size. Also, the more data there is to choose from, the more cherry picking is possible.

Decomposable in units of work As we will set the objective to only rebuild parts of the application, it has to be able to be decomposed in bite-sized units of work.

4.1.2 Selected case

The application that was selected to be the case in the case study is the Ordina Jobportal (2006), built on the Smart-Java development infrastructure. The primary function of this application is 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, et cetera. 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.

Conforming to reference architecture The Jobportal is one of the applications whose architecture has contributed to the reference architecture as it is now. Of course, there are some differences in implementation choices but the Jobportal architecture does fall within the domain of the reference architecture.

No toy application The Jobportal satisfies real user requirements: it is specifically designed to support the Recruitment personnel in their daily work and is actively used.

In production use (maturity) The application has been running in production since 2006, has undergone various maintenance steps and is still in evolution.

Not small The Jobportal has a total size 14.5k Non Commenting Source Statements (NCSS) and has a total 23 implemented use cases. We asses that this is more than enough to allow cherry picking for this study.

Decomposable in units of work The Jobportal followed the "RUP op maat" implementation of Rational Unified Process (Collaris and Dekker, 2008) and was, as RUP recommends, functionally decomposed into use cases. Use cases are described by Cockburn (2000) as follows:

> A use case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal.

The use cases have no interrelations and are considered to be well scoped units of work.

4.2 Case rebuilding

In this section, we carry out the second step in our research method (figure 4) and will discuss how we will rebuild the functionality of the Jobportal.

- First, we will lay down the rebuild goals and rules.
- As the Jobportal is a quite large application, we assess that it is not feasible to rebuild the entire application within the time constraint of this study. Yet, we want to do as much observations as possible in the time we have and will discuss how we selected the parts to rebuilt in order to maximize the observations done.

4.2.1 Rules for rebuilding

The primary goal is to use mod4j to completely clone the functionality within the scope of an use case and take record of any issues encountered in this process.

During implementation we have encountered situations where completing the functionality would no longer yield observations on working with mod4j. A very straightforward example is that mod4j currently does not support modeling the user interface: it does not contribute to the rebuild goal to clone the user interface as it is not supported by mod4j. For these situations, where mod4j does not influence the implementation at all, we will develop a proof of concept to verify the functionality can indeed be invoked and placed within an appropriate extension point, but will not invest the time to implement it fully. This does impact the validity of the research, as we did not ascertain that the proof of concept can be indeed implemented, however, we will address this in section 4.3.

4.2.2 Use case selection

Sampling technique The goal of the use case sampling is to maximize the number of insights on working with mod4j. The sampling technique we will use to accomplish this is called *snowball sampling*.

> Snowball sampling is a method for selecting cases in a case study where the cases are picked one by one. The first case is examined and based on the findings of this examination the next is selected, and so on. This method can be used if the researcher is unfamiliar in the research terrain and / or if the researcher cannot reliably predict the findings of a case examination (Verschuren and Doorewaard, 2005).

In our case, *snowball sampling* can be used to select use cases one by one allowing new insights during the implementation to dictate the choice for a next use case. This approach is especially useful since there is much data to choose from and initially, the researcher will not be familiar with both the Jobportal and mod4j.

Selected use cases The first selected use case is a read-only use case of a user searching 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. For this first use case, a portion of the domain model had to be modeled, which could be reused in subsequent use cases.

Next, we chose to rebuild a use case of a recruiter maintaining his own vacancies, using a part of the domain model that was already built in the previous use case. As this use case creates, reads, updates and deletes data, it was expected to offer new insights into modeling data modifications using mod4j.

Finally, we chose a use case that maintained reference data in the Jobportal. 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 already implemented domain model. The rationale for this was that it would yield information on how domain model partitions could be integrated.

4.3 Validation

The third step in our research method (figure 4) is the validation of the produced data by means of testing the correctness, completeness and discussing the implementation with mod4j experts.

4.3.1 Correctness

In order to make it plausible that the mod4j implementation can satisfy the exact same requirements we have used the original implementation as specification. This means we have tried to use the original Jobportal datamodel as-is and let the mod4j service layer expose exact the same methods as the original. We will further determine the equality of by comparing the run-time results of invoking each operation in the service definition of the original and the mod4j implementation using an automated integration test suite and extensive test data. These tests will compare characteristics like the number of results and the contents of the properties and associations of the returned objects. By validating that each exposed piece of business functionality yields the *exact* same results, it is made plausible that the mod4j implementation has satisfied the functional requirements.

4.3.2 Completeness

The implementation might not be complete due to issues with the test data, accidental omissions or errors. To this end a coverage tool will be used to verify that indeed all code in the current implementation has been covered. If this is the case, we have made it plausible that all functionality of the original implementation has been duplicated. Note that this validation also makes it plausible that the used test data is extensive enough.

4.3.3 Walkthrough

Another threat to the validity of the data is that mod4j can be used in many ways and we want to validate that we have used mod4j optimally. Any suboptimal usage can cause lost time, and, if not discovered at all, cause invalid conclusions and recommendations.

We have addressed this threat by sharing the source code of the developed implementation with the mod4j team and organizing a walkthrough to discuss it. A walkthrough is similar to a code inspection:

> In a walkthrough, a group of developers—with three or four being an optimal number—performs the review. Only one of the participants is the author of the program. Therefore, the majority of program testing is conducted by people other than

the author, which follows the testing principle stating that an individual is usually ineffective in testing his or her own program (page 23, Myers, 2004).

In code inspections, there is usually no discussion on how errors could be fixed in order to use time most efficiently. For this walkthrough, this principle was not employed as finding errors is already addressed by the aforementioned tests. Instead, the focus of this meeting was to discuss the use of the mod4j models and the extension points. Also, the proof of concepts mentioned in the previous section were discussed extensively to determine their suitability.

The identified issues during the walkthrough were fixed in the implementation but also kept record of for further analysis. It is an opportunity to analyze these issues and recommend how they might be prevented in future use.

4.4 Criterion evaluation

Now that we have rebuilt several use cases of the Jobportal, we can discuss the final step in the research method (figure 4): we will lay down how we will analyze the research data in order to reach a verdict regarding the fulfillment of each criterion (section 3).

4.4.1 Conformance to the reference architecture

In order to determine if an application built with mod4j conforms to the reference architecture, the following artifacts will be analyzed:

1. The source code of implemented application.
2. The list of encountered issues during implementation and walkthrough.

The conformance to the reference architecture will be determined by first extracting requirements from the reference architecture documentation (Boxtel, Malotaux and Tjon-a Hen, 2008). In this document, the requirements have been laid down in an itemized, concise manner and therefore the conformance to them can be determined well. We present an example:

- Domain objects must keep their internal state consistent.

We consider this requirement to be completely fulfilled: mod4j allows validation rules to be specified for attributes, such as maximum length, allowed to be empty, et cetera. Each of these rules is fired each time an attribute is changed, keeping the internal state of the domain object consistent. If required, developers can add their own custom business rules, adding hand-written code to the generated business rule stub[6].

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

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 or implementing code in the extension points.

Not at all The architectural requirement is not addressed at all in the generated code, but developers are able add hand-written code at a suitable extension point to fulfill this requirement. A boundary condition is that manually implementing the requirement is feasible, else the requirement is considered *violated*. For example, it is not feasible when large parts of the generated code or configuration need to be manually duplicated in order to fulfill the requirement.

Partial fulfillment The architectural requirement is not completely fulfilled: the rationale of the requirement is present but something is missing. This may occur when a single requirement defines several related rules of which some are fulfilled but others are not. The rationale of separating these requirements from the not fulfilled requirements is that it is typically less effort to change mod4j to fulfill the partially fulfilled requirement.

Complete fulfillment The architectural requirement is completely fulfilled by the code generated by mod4j. Note that this label is not a guarantee that an application using mod4j will always fulfill this: any hand-written code or configuration can still violate the requirement.

[6] A stub contains a temporary substitute for yet-to-be-developed code.

4.4.2 Real-life functional requirement satisfaction

Observations regarding the fulfillment of functional requirements are also elicited from the mod4j implementation itself and the list of issues encountered during implementation. The goal of the analysis will be to identify the cause or causes that certain functionality could not at all, not easily or not completely be implemented. This will be accomplished by selecting the functional, high priority issues from the list of issues and determining the root cause of each issue.

4.4.3 Reduction of hand-written code

The goal of the analysis of the amount of hand-written code is two-fold: first, to determine if in fact the amount of hand-written code has decreased compared to the original implementation and second, to identify if there is any manual code that is could be generated by the current DSLs. In order to do a valid comparison, we have focused only on functionality that has been completely duplicated using mod4j: that what is invoked in the correctness and completeness tests.

We will use the covered byte code statistics as metric to determine the amount of hand-written code. The statistics are created by the free Eclipe plugin EclEmma, a coverage tool that instruments byte code to show which code has been actually invoked by a certain execution. Code coverage statistics are often used during unit testing to determine if all code is tested (Zhu, Hall and May, 1997). 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.

5 Research results

In this section, we will answer the question to what degree mod4j meets the established criteria (section 1.3, second research question). For each criterion, we will:

- Present, outline and discuss the research data.
- Present the observations done:
  - Outline the observation.
  - Determine the cause of the observation, grounded by the research data.
  - Assess the severity or priority of the observation. The severity or priority is grounded by the research data and is used to determine if the cause of the observation threatens the suitability of mod4j on the specific criterion.

5.1 Conformance to the reference architecture

5.1.1 Research data

The complete list of requirements research data is quite long (72 requirements) and therefore has not been inlined here and can instead be found in Appendix A. In these tables we provide a complete list of the requirements harvested from the architecture document (Boxtel, Malotaux and Tjon-a Hen, 2008). Next, we set out to select the important requirements by categorizing the data based on whether the requirement states if a fact *must*, *should* or *may* be true. We present three examples of requirements found in the architecture document:

- Domain objects *must* keep their internal state consitent (must or mandatory requirement)
- Data Service agents *should* encapsulate access to just one service (should-requirement)
- Domain objects *may* broadcast events about change in state (may or optional requirement)

For each requirement, we determined which type it is, to what extent mod4j fulfills the requirement and a short rationale on how we reached this verdict. Next, we enriched the table in Appendix A with this information. The diagram in figure 5 provides a graphical view of this. Note that in this view, the requirements have *not* been weighted or prioritized and is therefore not suitable to use to draw quantitative conclusions from. However, it does show that interesting observations can be expected, since mod4j does not fulfill all architectural requirements.

We argue that completely fulfilled architectural requirements and optional (may) requirements will yield no observations, as long as they can be hand-written

when required (e.g, not violated). To filter out these requirements, we apply the filter in table 1.

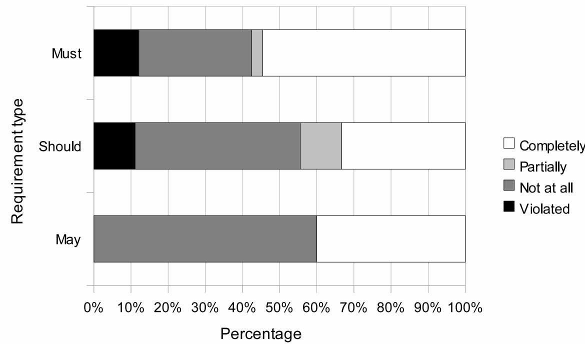


Fig. 5: Requirement fulfillment statistics

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

Tab. 1: Requirements filter

The √ symbol indicates that the requirement is relevant for further analysis, the x character indicates the requirement is filtered out.

Applying this filter has resulted in the following list of observations:

1. The business processes follow a different nomenclature (Appendix A, requirements 16, 27, 20-22)
2. There is no modeling support for business workflows (Appendix A, requirements 23-26)
3. There is no modeling support for service agents (Appendix A, requirements 27-33)
4. A *data access logic component* is generated for every domain object instead of only for high-level domain objects (Appendix A, requirement 61)
5. There is no modeling support for data service agents (Appendix A, requirements 42, 44-48)
6. Local services have no notion of security (Appendix A, requirements 2, 63-68, 70, 71)
7. Data access logic components do not support paging facilities for large amounts of data (Appendix A, requirement 39)
8. Hand-written code and configuration in extension points can violate architectural requirements. (Appendix A, all requirements)

In the following eight sections, we will discuss each observation in detail.

5.1.2 Observation 1: business processes nomenclature

Outline Business processes are defined by the reference architecture as short-lived, synchronous business functionality initiated by a single actor. Each operation is an atomic action: it either succeeds or it does not and no intermediary state is persisted. Mod4j does not generate anything by the name *business process*.

Although the nomenclature is not followed, business processes are supported. Mod4j follows the Domain-Driven Design nomenclature for this component and names these *domain services* (Evans, 2003). These domain services are partially generated by mod4j using the service definition laid down in the Service DSL (see figure 3) and can be extended with hand-written functionality. The generated methods follow the architectural guidelines on business processes: they are invoked by a direct call from one actor, they only invoke methods on *data access logic components* or domain objects, execute synchronously and in a single transaction.

Cause We cannot determine a cause for the difference in nomenclature based on the research data we have.

Severity As the studied case indicates that the requirement business logic functionality can be placed in the business process, the architectural guidelines are followed and only the name is different, we consider this to be not an issue.

5.1.3 Observation 2: no modeling support for business work-flows

Outline Business work-flows are defined as a series of possibly asynchronous invocations of business processes, service agents or other business work-flows. The business work-flows cannot be modeled in mod4j.

Cause The reference architecture mentions that there is insufficient experience on how business work-flows should be implemented. The lack of experience indicates it is not a common concern of projects in Smart-Java and although it is mentioned in the reference architecture it cannot be that important to the success of mod4j. Therefore, in the study we will place the implementation of business work-flows out of scope.

Business workflows can be implemented using hand-written code and manually integrated with the service layer, but mod4j will offer no support whatsoever for this.

Severity If an application that does require business work-flows and service agents this will mean that mod4j and possibly the entire reference architecture may not (yet) be suitable to implement this. While implementing the studied case, we did not encounter any problems related to the lack of business workflow support (Appendix B). It is important to bare in mind for the future, but for now, the experiences with the studied case do not indicate it imposes a limitation on the suitability of mod4j.

5.1.4 Observation 3: no modeling support for service agents

Outline Service agents execute external business logic (ie. an external credit card authorization service) and can be invoked only by business work-flows. However, there is no support for this in mod4j.

Cause It is described in detail how service agents should be implemented, however, they are only allowed be invoked by business work-flows. As these business processes are not supported (see previous section), it is to be expected that the service agents are not as well. Service agents can be hand-coded and integrated with - also hand-written - business workflows.

Severity Due to the nature of these service agents, we regard it is unlikely that more then a few of these service agents are present per system and that their technical implementation may vary since they might have to integrate with all sorts of systems via various protocols. Apart from the fact the rebuilt case did not require service agents (Appendix B), we do not have any further data to ground this claim. Yet, based on our own experiences with the Jobportal, we do not consider this omission to be a threat for the suitability of mod4j.

5.1.5 Observation 4: data logic access components for high-level domain objects only

Outline Mod4j generates a *data access logic component* for every modeled domain object. This directly violates the requirement that only high-level domain objects should have a *data access logic component*. We will first discuss the technical context of the issue and then determine the cause mod4j behaves in this manner.

Context The reason the reference architecture recommends distinguishing between high-level and regular domain objects is to split the domain model up in highly cohesive partitions that have low coupling to the rest of the domain model. The aggregate is introduced as a pattern in Domain-Driven Design (DDD) (Chapter 6, Evans, 2003) and is discussed in the context of the domain model, but in essence it follows the guidelines of Parnas (1972) for system modularization. An example is given in figure 6, the *OrderLine* and *OrderHistory* can only be retrieved through the aggregate root - the *Order*. Also, the associations that cross the aggregate boundary are uni-directional: the *OrderLine* can retrieve this Item, but the *Item* is not allowed to reference back to the *OrderLine*. The aggregate root can be retrieved through a *data logic access component* (called *Repository* in DDD). The objects within the aggregate boundary can only be retrieved through their parent and are not allowed to have an own *Repository*.

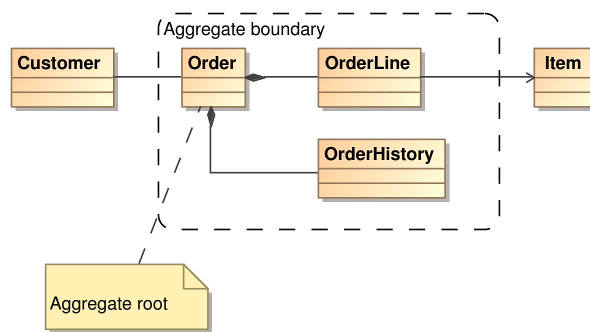


Fig. 6: Aggregate root (UML2 class diagram)

Cause Because the BusinessDomain DSL does not allow to define an aggregate and its root, mod4j cannot make the distinction between a high-level and normal domain object. Because this distinction cannot be made, mod4j treats every domain object as a high level object. This prevents modularization of the domain model, which, according to Evans, complicates maintenance. In mod4j, this causes superfluous *data access logic components* which cannot be removed by any means; they will always be generated even if they are never (allowed to be) 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.

Severity As we can only assess the severity based on the experiences rebuilding the Jobportal, we feed-forward to the next section (5.2), where we will elaborate on an issue (number 18) that could be resolved by means of an aggregate. Based on this, we conclude that the omission of the aggregate is indeed a problem in real-life projects and therefore regard this observation to be severe.

5.1.6 Observation 5: no modeling support for data service agents

Outline There is no modeling support for Data Service agents in mod4j. Data Service agents are described by the reference architecture to be used for retrieving and persisting domain objects managed by another systems. A typical application of a Data service agent is to retrieve a *Customer* domain object from a service exposed by a Customer Relationship Management (CRM) system. The Data Service agent shields the *data access logic component* of the technical details of the service implementation.

Cause Data Service agents are components that allow the integration of a system with other subsystems, just like the Service agents (observation 3). The reference architecture assumes that must domain objects are persisted through ORM and that the Data Service agents are not used very often (Chapter 3, Boxtel, Malotaux and Tjon-a Hen, 2008). The added value of generation these components will therefore probably be low. Depending on the environment of the application, the implementation of Data Service agent can also vary greatly, making it even harder to generate them.

The Data service agents can be manually implemented by overriding the generated *data access logic component* to invoke a hand-written Data service agent. Yet, it would be preferable if mod4j does either not generate the component and configuration at all or generates a stub that has to be manually implemented. This prevents dead code and configuration in the application.

Severity As Data service agents are not considered to be used more than occasionally within a system and they can be manually implemented. As no issues regarding Data service agents occurred during the rebuilding of the Jobportal (Appendix B), we do not foresee this issue limiting the suitability of mod4j for use in a product line.

5.1.7 Observation 6: the service interface has no notion of security

Outline The mod4j Service DSL has no notion of security at all. However, the reference architecture lays down detailed guidelines on concerns in the service interface. The reason is applications in a SOA environment fall within its domain. The primary function of the service layer is described to be to isolate the logical business layer from any implications of the protocol used to remotely be invoked. Example of such protocols are SOAP over HTTP and MQ, RESTful Web Services, Enterprise Java Beans (EJB) and Remote Method Invocation (RMI) (Hohpe and Woolf, 2003). In the case of remote invocations a trust boundary is crossed when the

service layer is entered and therefore, it is a concern of the service layer to do authorization and authentication.

Cause The fact that the Service DSL does not address any security concerns is caused by the fact that currently, only local services are generated: the methods can only be invoked from within the same Java Virtual Machine (JVM) and no trust boundary is passed. If the calling party, often the presentation layer, has already authenticated and authorized the user, the local service layer does not need to re-verify this. We conclude that for centralized deployment, the missing notion of security does not impose any limitations.

Suppose mod4j is used to implement a SOAP Web Service: only the local service definition would not suffice now. This concern can then be addressed by another component that makes the service remotely invoke-able (see figure 7) and is integrated with the local service definition. Another option is to address the security concern by means of Aspect Oriented Programming (Visser, 2007). We conclude that when a mod4j application is deployed distributed, the security concern can be addressed by other components that integrate with the mod4j local service.

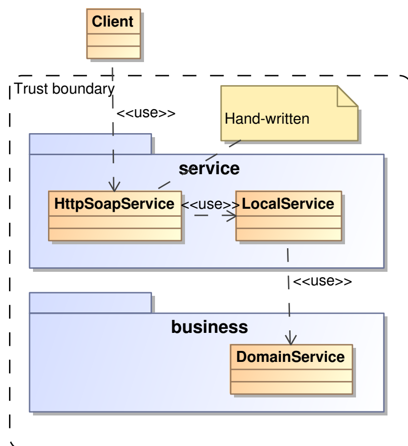


Fig. 7: Using mod4j for a SOAP Web Service (UML2 class diagram)

Severity Mod4j applications can be used both centralized and distributed by hand-coding a component that integrates with mod4j. Because the rebuilt case is not deployed distributed and all security concerns are addressed by the presentation layer, we do not consider this issue to be severe. We do underline that if the rebuilt case would have been deployed distributed, this might have caused a lot of hand-written code.

5.1.8 Observation 7: paging facilities not supported

Outline The out-of-the-box methods that return list of results, list-all and find, could possible return a very large data set. The reference architecture therefore prescribes that it must be possible to *page* through these results instead of returning the entire data set at once.

Cause The functional requirements of paging facilities are not always clear. For example, sorting results is a concern most placed at the client. What if a user sorts page of data? Should the query be refired with the sort column as argument? Or should the current returned value be sorted? This differs from application to application. Therefore, the choice was made by the mod4j team to not offer any paging facilities, but allow the developer to implement own methods that do support paging in the *data access logic components*.

Also, it is not likely that all the *data access logic components* will require paging functionality. The rebuilt case confirms this: there is no pagination support in the Jobportal.

Severity Since the functional requirements of paging are unclear and the rebuilt case indicates this functionality is indeed not commonly needed and it can be handwritten if required, we do not consider this issue to be a threat to the suitability of mod4j for Smart-Java projects.

5.1.9 Observation 8: hand-written code and configuration

Outline So far, it has been analyzed to what extent the generated code conforms to the reference architecture. However, code and configuration added manually in extension points is neither limited nor checked, which makes it possible to introduce code and configuration that violates architectural requirements. For example,

business logic can be entered in the local service extension point without triggering any warnings.

Cause If the code and configuration are limited by mod4j, a lot of flexibility will be lost. As we already identified, some variable features in the architecture are not addressed by mod4j (for example, service agents). Extension points allow these features to be hand-written. Only when all variable features can be addressed in models and practice has indicated that in fact extension points are no longer used can these be removed. Until then, extension points are a necessary evil.

Severity Imposing limitations on the manual code and configuration is not desirable, therefore, we consider this not to be an issue at all.

5.1.10 Conclusion

First, we provide an overview of the determined severities in table 2.

We have seen that mod4j follows the major parts of the reference architecture, but lacks modeling support for variable features (table 2, causes 2, 3, 5, 6 and 7). These variable features can be implemented by hand. This is in line with the goal of mod4j to deliver its added value in modeling parts that are used very often and allowing developers to extend functonality in extension points.

However, for the issue we consider to be severe (table 2, cause 4), we foresee difficulties in using mod4j to build applications conforming to the reference architecture.

5.2 Real-life functional requirement satisfaction

5.2.1 Research data

We present the encountered functional issues in table 3. They have been selected from the full record of issues encountered during implementation (Appendix B) by the following criteria:

- The issue describes a functional limitation or inability, marked in Appendix B by type *functional*.
- Severity is *major* or *blocker*. Major indicates a functional limitation, requiring hand-written code to circumvent or override generated code. Blocker indicates that the issue prevents the satisfaction of a functional requirement completely.

We have grouped closely related issues, resulting in the following observations:

1. Domain object persistency cannot be customized or disabled (issues 1, 2, 14, 15, 18)
2. Superfluous generated operations in the local service definition (issue 17)
3. Binary data is not supported (issue 4)

We will discuss these observations in the aforementioned order.

5.2.2 Observation 1: domain object persistency cannot be customized or disabled

Outline We have seen that a various issues relate to domain objects and their persistency (issues 1, 2, 14, 15, 18). It is deduced from the issues that mod4j does not offer a way to disable or overwrite the generated persistency functionality. In order to explain the cause of this, we will first provide some context on domain object persistency.

Context In the more legacy environments, it is not uncommon that a single data model is used by multiple applications. In these situations, the data model can be quite different from the domain model and the data and domain model should be strictly separated to prevent changes in the data model to ripple through to the domain service and vice-versa (Evans, 2003; Fowler, 2002). The mapping logic can be so complex that it is not feasible to use a object-relational mapper. Instead, the Data Mapper pattern (Fowler, 2002) can be used to write the complex mapping between the data and domain model by hand. This is a very time-intensive task that most projects try to avoid (O'Neil, 2008).

Due to the upcoming of service-oriented architectures, databases are shared with less peers and therefore are designed in a more domain-driven manner (O'Neil, 2008). The reference architecture assumes the use of object-relational mappers and in this case, the data

#	Observation cause	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 workflows 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	BusinessDomain DSL does not allowing 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.	-

Tab. 2: Conformance to the reference architecture summary

model is driven by the domain model (top-down) but influenced by the data model. A typical example is a version attribute in the domain object that enables optimistic concurrency control in the database[7] but has no functional meaning.

Cause Mod4j uses structural domain information laid down in the BusinessDomain DSL to generate both the domain model classes and the ORM configuration. Since any structural changes are only updated through the models, it is maintained on a single location. The domain and data model generation is split up to separate concerns, although they are both based on the same model. This way, the templates are more cohesive as they focus on the domain model or the persistency configuration but never both. However, the BusinessDomain DSL does not model concerns that are only related to persistency and therefore the generated configuration only contains data that is either generated for each class (static in the template) or relates only to the structure of the domain model. Additionally, mod4j allows the configuration of various properties in an application-wide configuration file that can influence aspects of the generation: for example, setting the strategy that is used to persist classes that inherit from other classes.

The encountered issues indicate that this does not yet suffice for production use.

Severity We consider this observation to be severe as the studied case indicates that the applicability of mod4j is limited: benefits of model-driven development are lost if generated configuration must duplicated, modified and maintained by hand. Also, mod4j will always keep generating the (possibly incorrect) configuration as this cannot be disabled.

5.2.3 Observation 2: superfluous generated local service operations

Outline As it was an objective to create an exact clone of the original implementation, the service definitions should match up for one hundred percent. However, comparing the service definitions of the mod4j and the original implementation, it was observed that the service definition generated by mod4j often offers more service methods than the original implementation and that this cannot be avoided.

Cause The Service DSL is used to define the service contract to the client. For every operation defined here, an implementation is generated in both the local and the domain service defines the operations top-down using DTOs. When a DTO is passed in the local service,

[7] Optimistic locking prevents corruption when users attempt to update the same data at the same time (Johnson, 2002)

#	Issue	Severity
1	Custom DAO implementation: cannot disable generation of code and configuration	Major
2	Cannot not override boolean persistency 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. Persistency 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

Tab. 3: Excerpt from Appendix B: functional issues, severity major or blocker

it may be based on an existing domain object. As DTOs may not be complete representations of domain objects - some properties might be omitted in the DTO - the local service needs to retrieve the domain object and copy every field in the DTO over to the domain object. This is the reason that the Service DSL demands that the read operation is defined in the service model. The side-effect is that the read operation is exposed in the local service as well, while it is not required or maybe not even functionally allowed.

We conclude that the superfluous operations are caused by the fact that the Service DSL can not differentiate between an operation for the local service and an operation for the domain service. When the local service requires a read operation to adapt a DTOs correctly, it can only demand that the operation is defined in the Service DSL and therefore in both service definitions.

If certain functionality is not allowed to be executed due to security considerations, the superfluous generated operation in the local service can be overridden in the local service extension point preventing it to be executed.

Severity This issue has been created because we wanted to rebuild an exact clone, but mod4j offered additional methods we neither needed nor could remove. We conclude that this issue is actually not driven by real-life functional requirements, but rather requirements we created when creating a rebuild strategy. Based on the research data, we cannot determine if the superfluous method are in fact a violation of the functional requirements of the Jobportal.

As this issue did not prevent functional requirements from being implemented in the studied case, we do not regard this issue to be severe.

5.2.4 Observation 3: binary data

Outline In the studied case, some domain model objects contained binary attributes, for example, a curriculum vitae (CV) object that is used to store the binary data of the document the applicant has provided. This functionality could not be implemented at all using mod4j without doing substantial, completely hand-written workaround.

Cause From the available research data, we cannot determine why the BusinessDomain DSL does not include support for binary data types. As blocking as the issue is for the studied case, as simple is it to solve this in mod4j: the ORM tool does support binary data.

Severity As the issue was blocking for the studied case, we regard this issue to be severe.

#	Observation cause	Severity
1	Persistency configuration is determined from the structural information laid down in the BusinessDomain 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.	+

Tab. 4: Real-life function requirements satisfaction summary

5.2.5 Conclusion

First, we have summarized the drawn conclusions on the severity of the identified observations in table 4.

We have rebuilt use cases of the Jobportal in order to ascertain if mod4j is in fact usable to implement the functional requirements of real-life applications. We found that for the rebuilt use cases mod4j does either support or allow the majority of functionality to be implemented.

However, we determined that two issues (table 4, causes 1 and 3) currently block the satisfaction of certain functional requirements in the Jobportal.

5.3 Reduction of hand-written code

5.3.1 Research data

The metric data of the covered code is provided in tables 5a through 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). The rationale for normalizing this is that our implementation is only a partial clone of the original Jobportal causes the Jobportal to have more initialization code that is unrelated to the use cases under test. The unrefined metric data is listed in Appendix C.

We will explain the context of the research data by elaboration on the design differences between the mod4j and original implementation first (figure 8a). The rationale that we discuss design differences is that we compare the original implementation with the mod4j implementation. In order to do valid comparison, we need to know where the design differs: this will help the analysis of observations.

Next, we will discuss how we interpreted the hand-written code statistics (figure 8b) in order to make the observations plausible.

Design differences We will first discuss figure 8a, a graphical view of table 5d (excluding totals).

- We see that mod4j requires 2663 bytecode 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 8a, data bar). The cause of the extra code in the data layer of the original implementation is that it has to adapt between behaviorless Transfer Objects used by the persistency 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 8a, domain bar). This is caused by the fact that mod4j domain objects combine the original BusinessObjects and TransferObjects[8] resulting in a decline of code in the mod4j domain layer compared to the orginal implementation.
- The original implementation has no executed instructions in the business layer at all (figure 8a,

[8] That are erroneously called ValueObjects in the current code (Fowler, 2002)

	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*

(a) UC02 Select Vacancy

	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*

(b) UC23 Maintain my vacancies

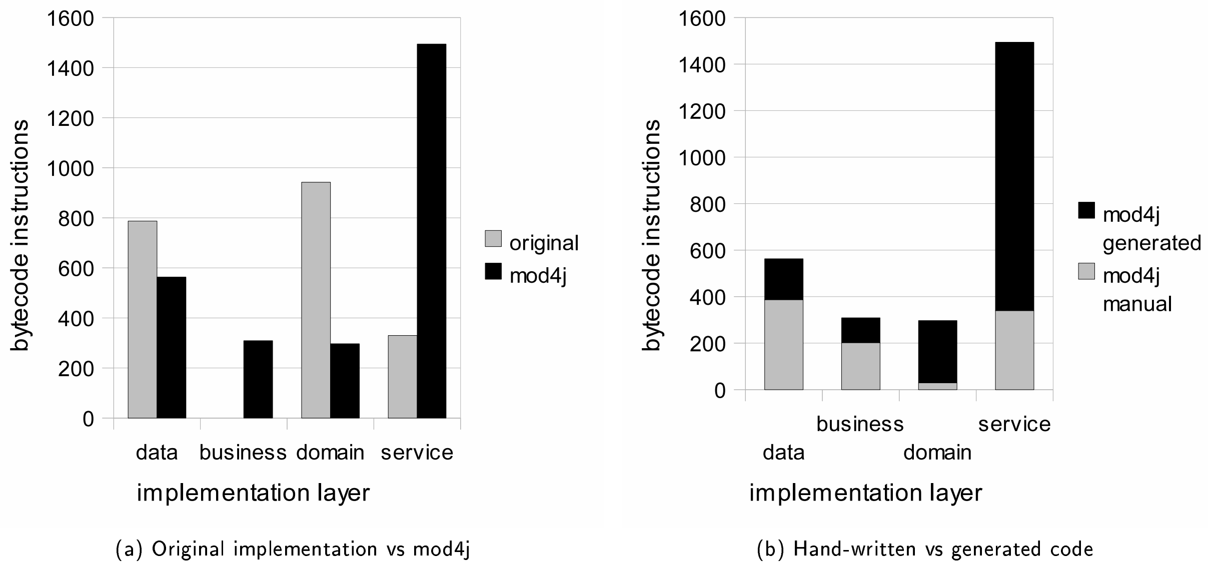
	original	mod4j manual	mod4j generated
data	339	116	65
business	0	56	53
domain	373	0	156
service	70	92	627
total	*782*	*264*	*901*

(c) UC11 Maintain reference data

	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*

(d) All three in one run

Tab. 5: Byte code instructions executed (*normalized*)



(a) Original implementation vs mod4j

(b) Hand-written vs generated code

Fig. 8: Hand-written code charts

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 8a, 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.
- Also, 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 8b (service bar).

Hand-written code vs generated code Figure 8bis 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* code, where the Jobportal code is all hand-written, totalling 2059 byte code instructions (table 5d). Based on these number, 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% (2059 -> 958), but we also identified in the previous section that the service layer has no real equivalent in the original implementation. The actual reduction is therefore even more: more then 50% of the *hand-written* byte code instructions is located in the service layer (542). 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 promising.
2. Of all the code in the analyzed sample, 64% is generated (table 5d). This excludes startup and intializing code as these have been subtracted during normalization. The unnormalized amounts, listed in Appendix C, indicate that 71% (table 9d) of the total code is generated.
3. The data layer has a large amount of hand-written code (~68%, data bar, figure 8b). Further analysis on the 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, and will discuss this in more detail (see observation 1 at the end of this section).
4. The domain layer is almost void of hand-written code (~10%, domain bar, figure 8b). 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 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 by specifying the rules in the Business-Domain DSL.
5. The business layer has a large amount of hand-written code (~66%: business bar, figure 8b). Closer inspection of the implementation showed

that some parts of this code pertained to boilerplate code that could have been generated (see observation 2 below). Also, the large amount of hand-written code is to be expected, as only the most simple business logic is generated. More complex business logic was implemented by hand.

6. The service layer has a considerate amount of hand-written code (~23%, service bar, figure 8b). Contrary to expectation, a lot of boilerplate code to invoke the domain service had to be handcoded. Since the concerns of the service layer could be well covered by the generators, we decide to analyze this in detail (see observation 2 below).

We re-iterate the observations we have decided to analyze in-depth:

1. The currently supported service methods do not not fully match what is required for the case under study (see item 3).
2. Hand-written boilerplate code in the local and domain service (see item 6).

We will discuss these observations in this order.

5.3.2 Observation 1: supported service methods

Outline A slight variant of the *findByExample* search method is implemented in four extension points, resulting in hand-written code that could be prevented if the method would be more flexible. It is also observed that counting the numbers of records returned for a certain query is done very often, which could be generated as well. These observations can be grouped: the currently supported service methods do not fully match what is required for the case under study.

Cause This is caused by the fact that the more specific service methods that are required cannot be generated by mod4j. This is specifically true where business process and work-flow logic is required as this can currently not be modeled in mod4j. But it is an opportunity to supporting application developers as much as possible by offering generic functionality. This also prevents a lot of boiler-plate code that is required to cross the various application layers.

Priority The percentage of hand-written code could be reduced by be up to 60% for the business layer and up to 68% for the data layer (figure 8b), depending on how much support for service methods will be added. We therefore consider this an important opportunity to explore.

5.3.3 Observation 2: hand-written boilerplate code

Outline It is observed that the service layer contains a non-trivial amount of hand-written code. The local service should by definition be a clone of the domain service, only shielding it from implementation details regarding the (remote) exposure to clients. In the case of the generated local service, this means it is only allowed do to a translation of DTOs into domain objects and vice-versa. Yet, we see that for every custom method we define in the Service DSL, hand-written code is required in the service layer.

Cause Mod4j recommends the use of custom DTOs, defined in the Datacontract DSL, to define arguments that are not a representation of one or more domain objects. An example of this is a getCustomerByName service method: as input argument the simple data type string suffices. A custom DTO can contain an arbitrary number of simple data types (string, double, et cetera) and enumerations, but domain objects are *not* allowed. As a result, the exact format of invocation to the next layer, the domain service, cannot be determined by mod4j: the different possibilities are listed in figure 9. We will elaborate on each of these options in the following items.

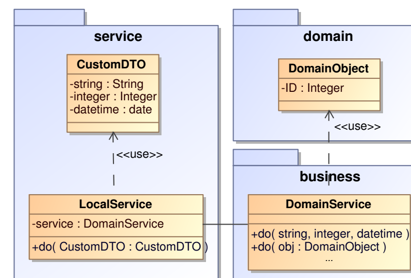


Fig. 9: Possible invocations in the domain service (UML2 class diagram)

- The primitive values that compose the custom DTO are used as arguments to invoke the domain service. This is the first operation shown in figure 9. This does not require the notion of a custom DTO: if the simple data types could be directly entered into the custom method declaration in the Service DSL, the domain service could be invoked with the same arguments: the local service can then be completely generated. We conclude that for this type of custom DTO usage, it is better to allow primitives to be entered in the custom method definition instead.

- The custom DTO might actually represent a single domain object (second operation in figure 9) but the Datacontract DSL may not be able to express this, as the DTOs in the datacontract that represent a domain object are not very flexible: only the inclusion of properties and associations from the domain object can be configured. It cannot, for instance, use a different name for a property. This is, however, an indication that the service contract exposed to the outside world and the internal domain model are not well compatible. Of course, this can occur for a number of reasons, yet, it is something that is best isolated from the rest of the system, for example using an *anti-corruption layer (Evans, 2003)*. A well-known examples of this is a XSL transformation of an incompatible service message. We conclude that there are better alternatives for this type of custom DTO usage.

- The custom DTO might be used for bulk data operations, such as for mass updates or report generation. Warmer (2007) calls these *ViewDTOs* and Fowler (2002) *RecordSets*: a direct, data-driven view of the underlying persistent storage. For these kind of operations, usage of the domain model and ORM is typically not well suited. Fowler (2002) recommends implementing a separate *data access logic component* with plain SQL directly populating the DTOs from this. The custom DTOs are not well suitable to this end: first, they live in the service layer and cannot be used in the data layer and second, they do not support data types that one would typically use to transfer bulk data: arrays, maps and lists. We conclude that the custom DTO is not suitable for the transfer of bulk data, but we do recognize that this is an omission in mod4j.

In conclusion, we can not determine the cause of why primitives are not allowed to be directly used in the method definitions and also which problem exactly is solved by the custom DTOs.

Priority Since it seems that the custom DTO is not required and its existence causes a lot of hand-written code (up 23% in the service layer, figure 8b), we think this is an important opportunity to explore.

5.3.4 Conclusion

First, 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. We conclude that mod4j is able to decrease the amount of hand-written code and therefore expect that high-level goals such as developer velocity might be achieved.

However, we have identified two opportunities (table 6, causes 1 and 2) to decrease the amount of hand-written code and will recommend on how to exploit these opportunities.

6 Recommendations

In the previous section, we reached conclusions regarding the suitability of mod4j. We will now analyze the causes of the severe observations and determine if a suitable recommendations can be done to address the issues. This will fulfill the research goal to reach recommendations on improving the suitability of mod4j for use in a Smart-Java software factory (section 1.3, third research question).

First, we summarize the causes of the severe observations we identified the previous sections.

1. The BusinessDomain DSL does not allow the modeling of an aggregate and its root (section 5.1, observation 4) .

#	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 cannot 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.	+

Tab. 6: Reduction of hand-written code results

2. Persistency code and configuration is generated based on the structure layed down in the BusinessDomain DSL and cannot be influenced by the developer (section 5.2, observation 1).

3. There is no apparent cause for the lack of support for binary datatypes in the BusinessDomain DSL: it can be added without serious effort (section 5.2, observation 3).

4. Currently, mod4j defines only the most basic service operations. It may be an opportunity to support more methods, preventing hand-written implementations (section 5.3, observation 1).

5. 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 cannot 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 (section 5.3, observation 2).

We will discuss the recommendations to resolve these issues in the following five subsections in the aforementioned order.

6.1 Aggregate root

The fact that MDSD is practiced offers the opportunity to include the aggregate root pattern explicitly in the DSL, something that is not feasible in a general purpose language (GPL). It may also possible to implement it implicitly by using lower-level concepts that are currently lacking in the BusinessDomain DSL. We will discuss both possibilities.

Note that this will also have implications in the Service DSL: if not all domain objects are persistable, this data must be exchanged between the Service and the BusinessDomain DSL.

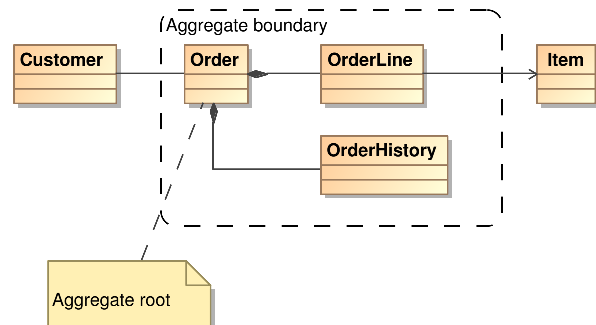


Fig. 10: Aggregate root (UML2 class diagram)

Explicit aggregate With explicit we mean that the BusinessDomain DSL itself will know the aggregate and its root as a concept. In the model, explicit keywords could indicate a class is an aggregate root and which classes are part of the aggregate. If we consider the example in figure 10, we could add the notion of an aggregate root to the DSL using the following concrete syntax (the *OrderHistory* and *Customer* are omitted here for brevity):

```
aggregate [
  root Order [ string name; ]
  class OrderLine [ string amount; ]
]

association Order order
            one <-> many
            OrderLine orderLines;

association OrderLine orderLine
// only uni-directional allowed!
            one -> one
            Item item;

class Item [ string description; ]
```

In this syntax, we expanded the language with an *aggregate* and *root* keyword. The *aggregate* keyword is used to model the aggregate boundary: it can have only one *root* and the other classes defined within it cannot be referenced from outside the boundary, for example, the association from *OrderLine* to *Item* is not allowed to be bidirectional.

Implicit aggregate With implicit we mean that the concept of aggregate root will not be an explicit part of the DSL but its building blocks will. This is similar to modeling an aggregate root in UML: often a stereotype (<<*AggregateRoot*>>) is used to model it (Greenfield et al., 2004; Stahl, Voelter and Czarnecki, 2006; Evans, 2003).

A key ingredient of the aggregate is the composite association which indicates that if the root is changed (or deleted), the associated objects are changed (or deleted) as well. Mod4j does not support the composite association, but if added, it can be used to both refrain from generating a *data access logic component* and configure a cascading delete for all aggregate children.

Part of the problem is now solved, however, a problem that remains is that not all objects within the aggregate have composite relations. Figure 11 shows a simplified example from the Domain Driven Design book (page 128, Evans, 2003). In this case, the *Position* object is part of the aggregate, but no direct composite relations reference it. In this case, all classes that reference a class that is referenced by a composite relation must be part of the aggregate boundary. This algorithm can become complex as the structure grows and, moreover, it will be less understandable for the developer using mod4j.

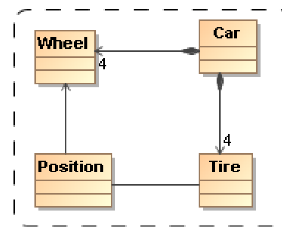


Fig. 11: Car aggregate root (UML2 Class diagram)

Conclusion As the implicit approach is more complicated both for mod4j and the developer, we recommend modifying mod4j to allow the modeling of the aggregate and its root explicitly. The proposed concrete syntax could be used as input to this.

6.2 Customized generated persistency functionality

We identify two distinctions in customizing the generated persistency functionality:

- No persistency: volatile domain objects that are never persisted.
- Tweaking persistency: customizing code or configuration pertaining to persistency.

We will discuss these items the following two paragraphs.

No persistency An issue we encountered was that we needed to model a list of *SearchResult*s (issue 15, Appendix B). A *SearchResult* has one attribute and one association: a score attribute which indicates how well the result matches the search query and an association to the found *Vacancy* domain object. While we could model this, we could not prevent the persistency functionality to be generated which is obviously undesirable in this case.

This could be resolved by splitting up the domain model in *Entities* and *Value Objects*. We will present a definition of these terms and then elaborate on how they can resolve the issue.

> A *Value Object* is a small, shareable and therefore immutable object, like money or a date range. Its equality isn't based on an identity (page 486, Fowler, 2002).

> An object defined primarily by its identity is called an *Entity* (page 93, Evans, 2003).

Evans (2003) describes in great detail how to model a domain by grouping *Entities* and *Value Objects* into aggregates (see previous section). One of the implications of a *Value Object* is that it is, on its own, not retrievable from the persistent storage since it has no identity. If *Value Objects* are persisted, it is always as part of an *Entity*. Hence, it is the concern of the *Repository* of the *Entity* to persist it. Fowler (2002) describes embedding the *Value Object* in the record of the *Entity* that references it (page 268).

The aforementioned *SearchResult* is an example of a *Value Object*: it has no identity and can be both immutable and shareable. Since the *SearchResult* only has an uni-directional association to the *Vacancy Entity* it is unambiguously not persistable. This is exactly what is required in the case of the *SearchResult*.

We propose the following concrete syntax for the BusinessDomain DSL:

```
valueobject SearchResult [ integer score; ]

association SearchResult searchResult
            one -> one
            Vacancy foundVacancy;

entity Vacancy [ ... ]
```

Here, the current keyword *class* of mod4j has been substitued with *entity*, as the *class* keyword is too generic: both an *Entity* and a *Value Object* are classes. Besides fulfilling the goal of having unpersistable domain classes, the split into *Entities* and *Value Objects* offers additional benefits: it prevents carrying around excess identity luggage when it is not required, allowing object sharing. The Fly-Weight Pattern (Gamma et al., 1995) elaborates on how small, sharable objects can be invaluable to performance.

Note that this will also have implications in the Service DSL: if some domain objects are no longer persistable, this data must be exchanged in the interface between the Service DSL and the BusinessDomain DSL.

Tweaking persistency configuration The persistency configuration is generated based on the structural data laid down in the BusinessDomain DSL and application-wide settings in a configuration file. The default strategy of persisting Java data types, for example a Java boolean as bit in the database, was not compatible with the Jobportal data model. This could not be resolved without duplicating the configuration and maintaining it by hand. Since this mapping configuration was consistent for the entire datamodel, the application-wide mod4j configuration should be extended to be able to customize the mapping of Java attribute types to database field types. Ideally, this should be done for each attribute type supported by both the ORM tool and mod4j[9].

6.3 Binary data types

As we did not identify any specific reason for excluding binary types in the BusinessDomain DSL, we recommend adding support for these datatypes in the BusinessDomain DSL and the artifacts it generates.

6.4 Supported service methods

We currently lack the research data (section 5.3, observation 1) to ground any recommendation on which service methods should be supported out-of-the-box. When analyzing the code of our studied case we see variance in required data access logic methods. The same applies for the issue we encountered regarding the flexibility of the find method (issue 3, Appendix B) : for one use case it needed to be more flexible but for other it didn't.

Also, we see that expanding mod4j with additional methods will create a scalability problem: the supported methods are currently hardcoded in the Service DSL and are always generated in the data layer. When expanding the supported methods, more flexibility is probably wanted here. To address the increasing complexity and footprint, we would like to point out the opportunity to add a new DataAccessLogic DSL that can be used to define the data access logic methods for a certain domain object. Since the Service DSL and the DataAccessLogic DSL could exchange an interface, the custom methods defined in the DataAccessLogic DSL can be referenced to from the Service DSL. This will prevent hand-written code that is currently required to invoke custom data access logic methods.

[9] See `http://www.java2s.com/Code/Java/Hibernate/JavaTypeVSHibernateType.htm`.

Because we cannot determine which methods should be added or removed based on our research data, we do not recommend to add or remove any from the currently supported functionality.

6.5 Custom DTOs

We argued that the custom DTO is currently not required if it is allowed to use simple data types (string, integer) directly in the service definition (section 5.3, observation 2). Therefore, we recommend that the notion of the custom DTO is removed from the Datacontract DSL and the Service DSL is modified to allow simple datatypes to be directly entered in the service method definitions, allowing the boilerplate code to invoke the domain service to be generated.

7 Threats to validity

In this section, we will discuss various threats to the validity of this research and how we have tried to minimize these threats.

Uncovered issues Because we have only rebuilt parts of the case under study due to time constraints, we can not know for certain that we have uncovered all important issues regarding functional and non-functional use. We have tried to address this threat by carefully selecting the case and by employing the *snowball sampling* technique to select the use cases the rebuild.

Selected criteria Of course, the selected criteria also affect the research validity as there are might be blind spots in our research method. Our focus has been on base criteria that make it *possible* to implement application using mod4j, but have had not much focus on the actual benefits. We did identify that mod4j succeeds in reducing the amount of hand-written code for the rebuilt functionality, but could not elaborate on the actual savings in terms of time and money. We have tried to address this threat in our criteria selection process: by using an exploratory interview and a more in-depth workshop to collect expert advice and allowing multiple iterations before solidifying the criteria we expect to have found the criteria that allow us to evaluate mod4j optimally.

Recommendation validity We cannot know for certain that the recommendations done will indeed resolve the problems. For example, when the concept of the aggregate is introduced in the BusinessDomain DSL we do not know for sure if this will in fact allow the modeling of all domain model aspects in our studied case. Our conclusion that mod4j is suitable for the development of Smart-Java projects when certain recommendations are followed might then be invalid. We have tried to address this threat by discussing our intermediary results and the reached recommendations with the mod4j experts.

Evolution As we identified early in the study, the lack of existing project data and time have made it infeasible to address the evolution concerns of mod4j, for example, how well the models can cope with changing requirements. We were unable to address this threat and consider the evolution aspect a blind spot in this study. Once there is real-life project data available, this is one of the areas that could be researched in future work.

8 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 (Warmer, 2007). 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 theresults 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.

We can compare the percentage of generated code, as Warmer writes that the first project was delivered within time and budget. The amount of generated code was 73%, we have seen similar amounts of generated code in our measurements (71%, table 5d).

WebDSL Visser (2007) 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 his paper is the first place intended as a case study in the development of DSLs.

Although the approach of generation 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 In his Master's Thesis, van Dijk (2009) 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 toy application developed by the researcher. Like mod4j, he uses the openArchitectureWare tooling to design the metamodels 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.

9 Conclusion

We have done an extensive evaluation of the suitability of mod4j for building applications within the reference architecture domain. By eliciting the criteria in a structured way, consulting both literature and experts, we have selected optimal criteria for evaluating mod4j. Next, we have set up a research method with a strong focus on data validity to evaluate these criteria. For the issues that were identified to threaten the suitability of mod4j, we have determined and outlined in detail how they could be resolved using well-known, proven design patterns. We conclude that when these recommendations are followed, mod4j is suitable to be used to build applications that fall within the domain of the Ordina J-Technologies reference architecture. Because our measurements showed that up to 71% of the code can be generated, we consider it probable that applications will be built in less time and with less effort.

In a scientific context, this study has contributed real-life evaluation data of model-driven software development applied in an industrial context. Also, we expect that the structured approach, the research method and the criteria we have outlined will help future evaluations of model-driven tooling.

References

Bass, Len, Clements, Paul and Kazman, Rick, Software Architecture in Practice, Second Edition. Addison-Wesley Professional, April 2003 ⟨URL: `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0321154959`⟩, ISBN 0321154959. 2

Beck, Kent and Andres, Cynthia, Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004, ISBN 0321278658. 12

Bennett, K. and Rajlich, V., Software Evolution: A Road Map. Software Maintenance, IEEE International Conference on, 0 2001, p. 4, ISSN 1063–6773. 12

Bézivin, Jean, On the unification power of models. Software and System Modeling, 4 2005:2, pp. 171–188. 3

Boxtel, Pieter van, Malotaux, Eric Jan and Hen, Philippe Tjon-a, Ordina Java Referentie Architectuur. Ordina J-Technologies, 2008, Versie 1.1. 11, 15, 16, 19

Breslav, Andrey, DSL development based on target meta-models. Using AST transformations for automating semantic analysis in a textual DSL framework. CoRR abs/0801.1219 2008. 4

Cockburn, Alistair, Prioritizing forces in software design. 1996, pp. 319–333, ISBN 0–201–895277. 5

Cockburn, Alistair, Writing Effective Use Cases. Addison-Wesley Professional, January 2000 ⟨URL: `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0201702258`⟩, ISBN 0201702258. 13

Collaris, R.A. and Dekker, E., RUP op maat. 2008. 13

Czarnecki, Krzysztof and Eisenecker, Ulrich, Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, June 2000 ⟨URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0201309777⟩, ISBN 0201309777. 4, 11

Deursen, Arie van et al., Domain-Specific Languages. DRAFT DRAFT Annotated Bibliography. DRAFT ACM SIGPLAN Notices. DRAFT, 2000 – Technical report. 3

Dijk, David van, Changeability in model driven web development. Master's thesis, University of Amsterdam, 2009. 33

Dijkstra, E. W., EWD 447: On the role of scientific thought. Selected Writings on Computing: A Personal Perspective, 1982, pp. 60–66 ⟨URL: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html⟩. 5

Evans, Eric, Domain-Driven Design: Tacking Complexity In the Heart of Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN 0321125215. 5, 17, 18, 19, 21, 28, 30, 31

Fagan, Michael, Design and code inspections to reduce errors in program development. 2002, pp. 575–607, ISBN 3–540–43081–4. 1

Fowler, Martin, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN 0321127420. 5, 7, 21, 24, 28, 30, 31

Fowler, Martin, Inversion of Control Containers and the Dependency Injection pattern. January 2004 ⟨URL: http://www.itu.dk/courses/VOP/E2005/VOP2005E/8\protect\T1\textunderscoreinjection.pdf⟩. 7

Fowler, Martin, Language Workbenches: The Killer-App for Domain Specific Languages? May 2005 ⟨URL: http://www.martinfowler.com/articles/languageWorkbench.html⟩. 4

Gamma, Erich et al., Design Patterns. Boston, MA: Addison-Wesley, January 1995 ⟨URL: http://www.amazon.co.uk/exec/obidos/ASIN/0201633612/citeulike-21⟩, ISBN 0201633612. 31

Greenfield, Jack and Short, Keith, Software factories: assembling applications with patterns, models, frameworks and tools. in: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. New York, NY, USA: ACM, 2003 ⟨URL: http://dx.doi.org/10.1145/949344.949348⟩, ISBN 1–58113–751–6, pp. 16–27. 2

Greenfield, Jack et al., Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, August 2004 ⟨URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0471202843⟩, ISBN 0471202843. 2, 11, 30

Hailpern, B. and Tarr, P., Model-driven development: the good, the bad, and the ugly. IBM Syst. J. 45 2006:3, pp. 451–461, ISSN 0018–8670. 2, 3, 9

Hohpe, Gregor and Woolf, Bobby, Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, October 2003 ⟨URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0321200683⟩, ISBN 0321200683. 19

Johnson, Rod, Expert One-on-One J2EE Design & Development. Birmingham, UK, UK: Wrox Press Ltd., 2002, ISBN 1861007841. 22

Kleppe, Anneke, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, 2008, ISBN 0321553454, 9780321553454. 3, 4

Myers, Glenford J., The Art of Software Testing, Second Edition. 2nd edition. Wiley, June 2004 ⟨URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0471469122⟩, ISBN 0471469122. 15

O'Neil, Elizabeth J., Object/relational mapping 2008: hibernate and the entity data model (edm). in: SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, 2008, ISBN 978–1–60558–102–6, pp. 1351–1356. 21

Parnas, D. L., On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM, 15 December 1972:12, pp. 1053–1058. 18

Spinellis, Diomidis, Notable design patterns for domain-specific languages. J. Syst. Softw. 56 February 2001:1, pp. 91–99 ⟨URL: `http://dx.doi.org/10.1016/S0164-1212(00)00089-3`⟩, ISSN 0164–1212. 4

Stahl, Thomas, Voelter, Markus and Czarnecki, Krzysztof, Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2006, ISBN 0470025700. 3, 4, 11, 30

Verschuren, P. and Doorewaard, H., Het ontwerpen van een onderzoek. Boom/Lemma, 2005. 14

Visser, Eelco, WebDSL: A Case Study in Domain-Specific Language Engineering. in: GTTSE. 2007, pp. 291–373. 20, 32

Vlissides, J., Generation Gap [software design pattern]. C++ Report, 8 November -December 1996:10, pp. 12, 14–18, ISSN 1040–6042. 4

Warmer, Jos, A Model Driven Software Factory Using Domain Specific Languages. in: ECMDA-FA. 2007, pp. 194–203. 9, 28, 32

Zhu, Hong, Hall, Patrick A. V. and May, John H. R., Software unit test coverage and adequacy. ACM Comput. Surv. 29 1997:4, pp. 366–427, ISSN 0360–0300. 16

A Architectural requirements

Tab. 7: Architectural requirements

#	Architectural requirement	Type	Fulfillment	Rationale
1	Service interfaces must do security checks	Must	Violated	Security concerns are not addressed
2	Service interfaces must expose "course grained" operations	Must	Complete	DTOs provide course grained arguments
3	Service interfaces may not contain business logic	Must	Complete	Generated code does not contain business logic.
4	Service interfaces may only call a domain service	Must	Complete	See previous requirement.
5	Service interfaces should shield the domain service of technical implications of the implementation (e.g. SOAP, RMI)	Should	Complete	Course-grained DTOs are shielded from domain model.
6	Service messages should not expose business logic to the outside world	Should	Complete	Messages are based on DTOs, which are adapted in business objects
7	Service messages should contain partial or complete representations of domain object(s)	Should	Partial	Normal DTOs do this, custom DTOs not!
8	Domain objects must keep their internal state consistent	Must	Complete	Done by business rules.
9	Domain objects expose operations to execute actions and calculations	Must	Partial	No behaviour in model
10	Domain objects must be are unaware of if and how they are persisted	Must	Complete	All persistency concerns generated into data layer
11	Domain objects may call Data Access Logic components (implicitly and explicitly)	May	Complete	Through lazy-loading of the ORM tool
12	Domain objects may delegate the validation of a business rule to a lower layer (such as the database)	May	Complete	i.e. unique-rules are delegated to the data model
13	Domain objects may broadcast events about change in state	May	Not at all	Not supported nor prevented
14	Business rules should throw exceptions when the state of a domain object is inconsistent	Should	Complete	Business rules throw exceptions
15	Business rules should return an error object (Special case pattern (Fowler, 2002)) when the error is specific to a certain invocation	Should	Violated	Does not fit in current business rule model
16	Business processes should implement logic specific to a single business process	Should	Not at all	Business process nomenclature
17	Business processes may call domain objects	May	Not at all	Business process nomenclature
18	Business processes take care of persisting changes by calling data access logic components	Must	Complete	Generated code does this.

Tab. 7: Architectural requirements

#	Architectural requirement	Type	Fulfillment	Rationale
19	Business processes may call data access logic components to retrieve domain objects from persistent storage	May	Complete	Generated code does this.
20	Business processes may implement custom transactions if technical transactions do not suffice	May	Not at all	Business process nomenclature
21	Business processes may send business process events (asynchronously) to external actors	May	Not at all	Business process nomenclature
22	Business processes may implement re-try behaviour for technical errors	May	Not at all	Business process nomenclature
23	Business workflows may invoke other workflows and processes	May	Not at all	Not supported nor prevented
24	Business workflows may invoke external services using service agents	May	Not at all	Workflows neither supported nor prevented
25	Business workflows should validate input before invoking the next process	Should	Not at all	Workflows neither supported nor prevented
26	Business workflows should not do direct invocations on domain objects or data logic components	Should	Not at all	Workflows neither supported nor prevented
27	Service agents execute business logic and do not facilitate data access (ie. credit card authorization service)	Must	Not at all	Service agents neither supported nor prevented
28	Service agents are usually not included in the scope of distributed transactions	Could	Not at all	Service agents neither supported nor prevented
29	Service agents shield business workflows from technical implications of the service implementation	Must	Not at all	Service agents neither supported nor prevented
30	Service agents should encapsulate access to just one service.	Should	Not at all	Service agents neither supported nor prevented
31	Service agents must provide input and output data formats that are compatible with the business components calling the service. In doing so, it isolates the business layer from the service implementation in terms of data format or schema changes. This rule implies that mapping between these formats is also a responsibility of a service agent.	Must	Not at all	Service agents neither supported nor prevented
32	Service agents must set the right security context or provide the right credentials to the service for authentication.	Must	Not at all	Service agents neither supported nor prevented
33	Service agents may cache results from service calls.	May	Not at all	Service agents neither supported nor prevented

Tab. 7: Architectural requirements

#	Architectural requirement	Type	Fulfillment	Rationale
34	Data access logic components must expose the so-called CRUD methods for inserting, deleting, updating and retrieving data	Must	Complete	Out-of-the-box generated for each DAO
35	Data access logic components may expose methods to do queries and return either a list of objects, an object graph, a list of object graphs or a view on persistent data	May	Complete	List and find are generated example, more methods can be added by hand
36	Data access logic components adapt the domain object to the format that is used to store the object	Must	Complete	Taken care of by the ORM tool
37	Data access logic components should support different data stores transparently	Should	Complete	Taken care of by the ORM tool
38	Data access logic components may implement a caching strategy	May	Complete	Taken care of by the ORM tool
39	Data access logic components have to support paging facilities for large amounts of data	Must	Not at all	Paging neither supported nor prevented
40	Data access logic components may decrypt/encrypt data	May	Not at all	Encrypt/decrypt neither supported nor prevented
41	Data access logic components should not invoke other data access logic components	Should	Complete	
42	Data access logic components may invoke data service agents	May	Not at all	Data service agents neither supported nor prevented
43	Data access logic components must be stateless	Must	Complete	
44	Data Service agents must isolate the data access logic components from the intricacies of the communication / transport protocol.	Must	Not at all	Data service agents neither supported nor prevented
45	Data Service agents should encapsulate access to just one service	Should	Not at all	Data service agents neither supported nor prevented
46	Data Service agents must provide input and output data formats that are compatible with the data access logic component calling the service. In doing so, it isolates the data layer from the service implementation in terms of data format or schema changes. This rule implies that mapping between these formats is also a responsibility of a service agent.	Must	Not at all	Data service agents neither supported nor prevented
47	Data Service agents must set the right security context or provide the right credentials to the service for authentication	Must	Not at all	Data service agents neither supported nor prevented
48	Data Service agents may cache results from service calls	May	Not at all	Data service agents neither supported nor prevented

Tab. 7: Architectural requirements

#	Architectural requirement	Type	Fulfillment	Rationale
49	The service layer may only reference other layers as specified in Figure 12	May	Complete	Generated code honors this.
50	The service layer must be implemented using an unique package prefix (e.g, nl.refapp.service)	Must	Complete	Generated code honors this.
51	The business layer should be implemented as Plain Old Java Objects (POJOs)	Should	Complete	Generated code honors this.
52	The business layer may only reference other layers as specified in Figure 12	Must	Complete	Generated code honors this.
53	The business layer must be implemented using an unique package prefix (e.g, nl.refapp.business)	Must	Complete	Generated code honors this.
54	The domain layer must be implemented using an unique package prefix (e.g, nl.refapp.domain)	Must	Complete	Generated code honors this.
55	The domain layer may only reference other layers as specified in Figure 12	Must	Complete	Generated code honors this.
56	The domain layer are implemented as POJOs	Must	Complete	Domain objects are POJOs (no extends / implements)
57	The domain layer must use the Spring Inversion of Control container for dependency management	Must	Complete	Configuration for Spring is generated
58	The data layer must be implemented using an unique package prefix (e.g, nl.refapp.data)	Must	Complete	Generated code honors this.
59	The data layer may only reference other layers as specified in Figure 12	Must	Complete	Generated code honors this.
60	It is recommended to use the Java Persistency API (JPA) for object/relational mapping	Should	Violated	Hibernate used instead
61	The data layer must provide a data access logic component for each high level domain object (aggregate root (Evans, 2003)).	Must	Violated	DAO generated for each domain object
62	Use LDAP-based JavaEE authentication wherever possible, only reverting to other ways of authentication when necessary	Should	Not at all	Security concerns are not addressed
63	The service interfaces must authenticate as well (...)	Must	Violated	Security concerns are not addressed
64	The other components in the business and data layer normally execute in the same context as the services interface and thus may rely on the services interface to have done the authentication.	May	Complete	Other components have no notion of security

Tab. 7: Architectural requirements

#	Architectural requirement	Type	Fulfillment	Rationale
65	Data stores should preferably authenticate the calling process, not the end-user. However, authorization or auditing rules may imply that impersonation be used.	Should	Partial	Security concerns are not addressed
66	Use role based authorization. This is more manageable then directly coupling rights to identities.	Must	Not at all	Security concerns are not addressed
67	Use declarative authorization where possible. Checking an identity in the code should be done seldomly.	Should	Not at all	Security concerns are not addressed
68	Service interfaces must authorize requests as well (...)	Must	Violated	Security concerns are not addressed
69	Other components in the business and data layer may rely on the authorization of the service interface.	May	Complete	Other components have no notion of security
70	In addition to the role based authorization of the service interfaces, business workflows, business processes, or business classes may, where applicable, execute complex business logic dependent authorization rules. An example is the business rule that "an account manager may only see orders of his own accounts".	May	Not at all	Security concerns are not addressed
71	Use of standard JavaEE capabilities for determining authorization of users is preferable over custom implementations	Must	Not at all	Security concerns are not addressed

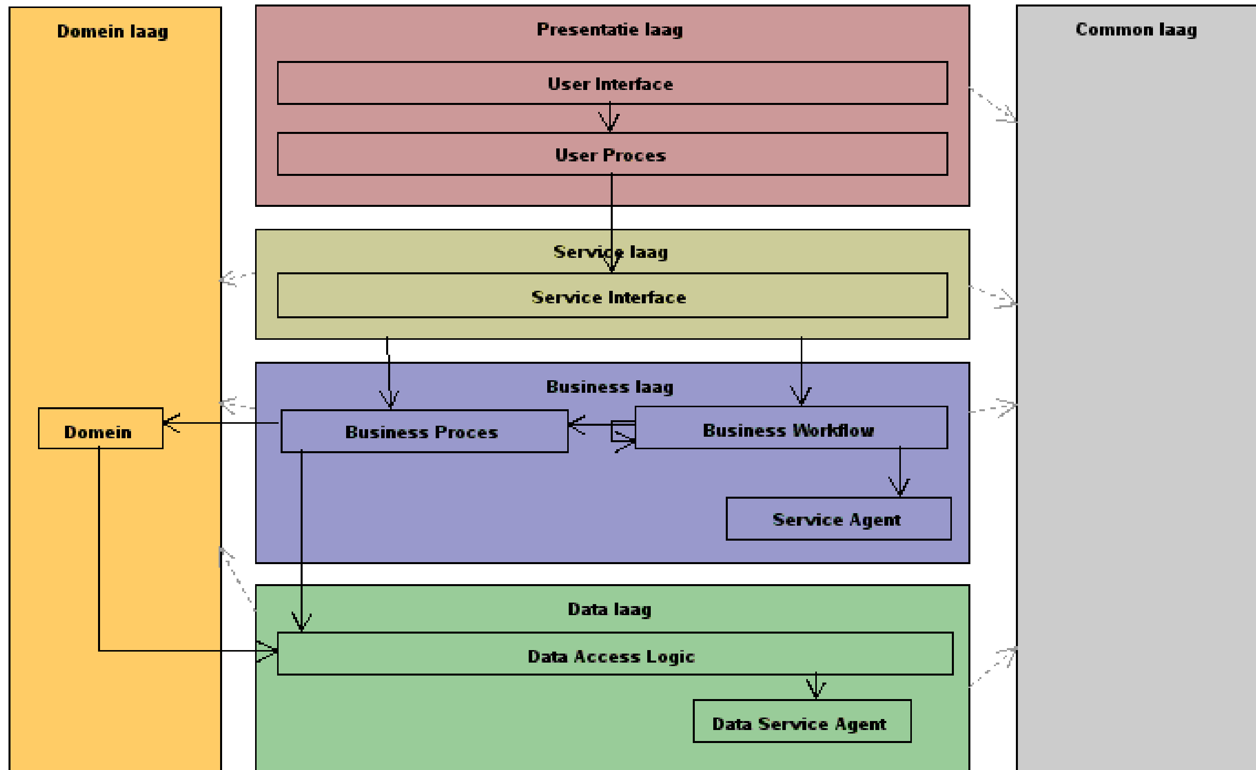


Fig. 12: Implementation view

B Functional implementation issues

Tab. 8: Functional issues

#	Issue	Type	Severity	Observation
1	Custom DAO implementation: cannot disable generation of code and configuration	Functional	Major	Domain object persistency cannot be customized or disabled
2	Cannot not override boolean persistency configuration	Functional	Blocker	Domain object persistency cannot be customized or disabled
3	The predefined methods find method is not flexible enough	Hand-written	Minor	Relates to hand-written code.
4	Binary data types are not supported	Functional	Blocker	No binary data type can be specified in the BusinessDomain DSL
5	Extending configuration can not be done in the same style (explicit vs implicit spring config)	Hand-written	Minor	Spring config
6	It's hard to extend spring configuration	Hand-written	Minor	Spring config

Tab. 8: Functional issues

#	Issue	Type	Severity	Observation
7	The DTO translators do not allow multiple DTOs to be translated at once, resulting in more manual code	Hand-written	Minor	Relates to hand-written code.
8	There is no easy way to test the models, especially the domain model. The only way is to manually bootstrap a domain model instance in Java. The DSLs should be more easily testable, by using a concrete syntax to concrete a model instance.	Hand-written	Wish	
9	When adapting a list of DTOs, a read operation is done for each separate DTO which is very performance intensive.	Performance	Minor	Adapting a list of DTOs
10	The local service contains hand-written business logic. This is a mistake of the developer.	Documentation	Minor	Extension point documentation
11	Documentation entered in the Service model is not applied to the generated Java code	Fixed	Minor	Fixed in next mod4j version
12	Unique constraints defined in mod4j have no effect if the data model is not generated by mod4j.	Functional	Minor	Should be made clear in documentation, but imposes no functional limitations as this concern can be addressed in the data model.
13	Associations in DTOs are not translated back to domain objects. This results in a more fine-grained service interface (add this, remove that, etc). A more coarse-grained service interface is preferable.	Fixed	Major	Fixed in next mod4j version
14	Mod4j generates incorrect ORM mappings if a domain object has a many-to-many association with itself. Workaround in place.	Functional	Major	This is a bug in mod4j which is being resolved. It is not blocking because a workaround is possible.
15	It is not possible to have a non-persistable domain object. Example: SearchResult. Persistency does not make sense here, yet mapping etc is generated.	Functional	Major	Domain object persistency cannot be customized or disabled
16	The local service could be Complete generated without extension points. By definition, it must duplicate the domain service. Now it causes a lot of hand-written code.	Hand-written	Major	Relates to hand-written code.
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.	Functional	Major	Superfluous operations in the local service definition

Tab. 8: Functional issues

#	Issue	Type	Severity	Observation
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).	Functional	Major	Domain object persistency cannot be customized or disabled
19	Referencing from business model to business model is broken.	Functional	Minor	Known defect, is being fixed.

C Hand-written code statistics

	original	mod4j manual	mod4j generated
data	410	259	213
business	0	158	120
domain	2406	46	874
service	296	189	745
total	*3112*	*652*	*1952*

(a) UC02 Select Vacancy

	original	mod4j manual	mod4j generated
data	395	300	222
business	0	22	112
domain	2319	71	892
service	115	85	632
total	*2829*	*478*	*1858*

(b) UC23 Maintain my vacancies

	original	mod4j manual	mod4j generated
data	403	260	185
business	0	73	142
domain	2234	42	929
service	143	105	839
total	*2780*	*480*	*2095*

(c) UC11 Maintain reference data

	original	mod4j manual	mod4j generated
data	851	531	296
business	0	219	196
domain	2803	71	1041
service	403	353	1366
total	*4057*	*1174*	*2899*

(d) All three in one run

Tab. 9: Byte code instructions executed (*not normalized*)