Introspective Model-Driven Development

Thomas Büchner and Florian Matthes

Chair of Software Engineering for Business Information Systems
Technische Universität München
Boltzmannstraße 3, 85748 Garching b. München
{buechner,matthes}@in.tum.de

Abstract. In this paper, we propose a new approach to model-driven development, which we call introspective model-driven development (IMDD). This approach relies heavily on some well-understood underlying abstractions, in order to bridge the abstraction gap between the requirements and the actual executable system. These abstractions are object-oriented programming languages and frameworks as a means of architectural abstraction. The main idea of IMDD is to annotate the extension points of a framework explicitly, which enables the automatic introspection of the defined metamodel. In a second step, a model of the customizations can be obtained by model introspection. There are two kinds of introspective frameworks – introspective blackbox and introspective whitebox frameworks. We developed an extension of the Eclipse IDE, which supports introspective model-driven development. Furthermore, we discuss the characteristics of the proposed approach, compared to established generative approaches.

1 Introduction

Dealing with the growing complexity of modern information systems is one of the challenges in computer science. One way to cope with this issue is the use of abstractions. There are some well-understood levels of abstraction as shown in figure 1.

The basic abstraction which hides some details of the underlying executable system is an object-oriented programming language (e.g. Java, C#). These languages are so-called *General Purpose Languages* (GPLs), which means that they are used to solve a broad spectrum of problems.

On top of object-oriented programming languages there are frameworks as a category of architectural abstraction. A framework embodies an abstract design for solutions to a family of related problems [1]. In order to solve a concrete problem, a framework has to be customized. The concrete task of customizing a framework involves the manipulation of low-level constructs like XML-files or code of the base programming language. The relationship between these constructs and the conceptual decisions in the problem space is not stated explicitly, and the intellectual distance between the adaptation constructs and the problem domain is pretty large.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 33–49, 2006.

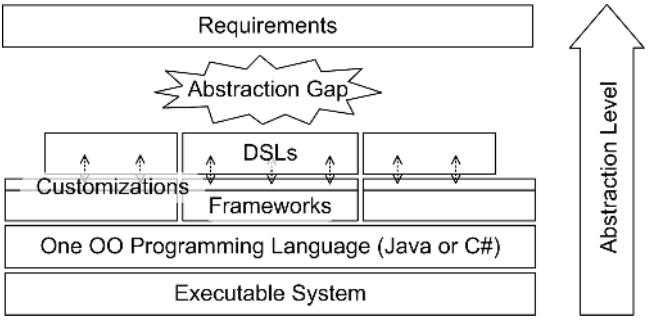

Fig. 1. Bridging the Abstraction Gap

One proposed solution to raise the level of abstraction of the framework customization process, is the use of a domain-specific language (DSL), which represents the extension points of a framewok in a usually declarative way. This approach is called model-driven development and implies an explicit connection between the high-level constructs of a DSL and the corresponding customization artifacts [3]. Technically speaking, there exists a transformation between the model and the customization artifacts. This distinguishes MDD from so-called *model-based* processes, in which models are merely used to illustrate certain aspects of a system in an understandable way, but the models created are not tied directly to the executable system. In this case, the models often do not "tell the truth" about the current system, and the creation of models is often seen as an overhead to the actual development process. So only with an MDD approach it is possible to obtain all benefits of using models to build information systems.

The most important point in using an MDD process is the explicit connection between the models and the actual customization artifacts. This leads to the question, in which direction the transformation is being applied. If the direction points from the model to the customization artifacts, this is called a *forward engineering* process [4]. Processes which use a transformation in the other direction are called *reverse engineering* processes. In this paper, we call these processes *top-down* and *bottom-up*.

A particular challenge for MDD processes results from the nature of the artifacts involved. In most cases, neither of them is sufficient to specify a complete system. Both, the model and the customization artifacts should be editable, and changes should lead to an immediate synchronization of the affected artifact. This is called *roundtrip engineering*. Realizing roundtrip engineering with a top-down process is a challenging task [5]. A promising approach to this problem is that of roundtrip visualizations [6].

All proposed implementations [7], [8] of model-driven development favor a top-down approach, in which they generate customization artifacts from models. To emphasize this, we call this approach *generative model-driven development*.

We propose in this paper a bottom-up approach, in which the high-level models are a rather transient result of an introspection process. We call this approach *introspective model-driven development*.

The article is structured as follows: In chapter 2 we give a short overview on generative model-driven development. In chapter 3, we introduce introspective model-driven development, which will be refined in chapter 4 and 5. In chapter 6, we will conclude with a comparison of the proposed approach with the prevailing approach to model-driven development.

2 Generative Model-Driven Development

An overview of generative model-driven development is illustrated schematically in figure 2. Similar to the life cycle of a framework the process is divided in a core development phase and an application development phase, with different roles of developers involved. The first result of the core development phase is the framework with its extension points. The creation of the core framework will be done by framework developers. In order to provide a more abstract view on the extension points of the framework, a language developer extracts the metamodel of the framework and creates a domain-specific language which reflects this metamodel. The extracted metamodel only reflects these parts of the framework, which will be customized in a declarative way. The metamodel will usually be specified using an existing meta-metamodel, as e.g., EMF [9] or MOF [10]. Given the metamodel, a *transformation developer* will create transformation rules which enable the transformation of models to concrete customization constructs. This will usually be done using a specific template language.

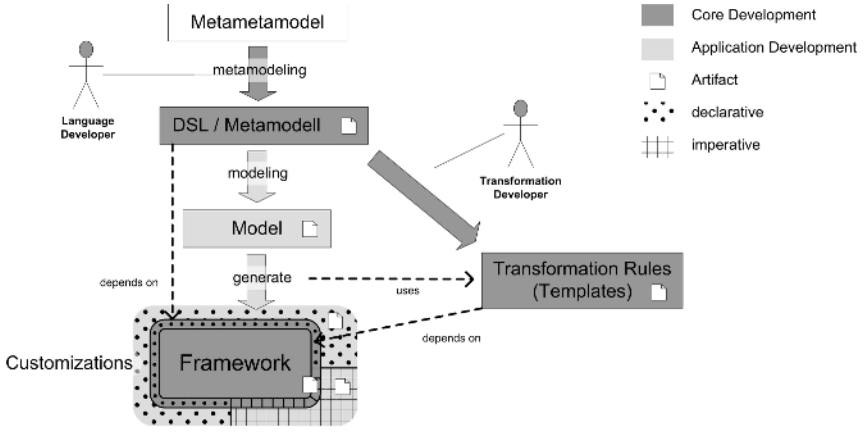

Fig. 2. Generative Model-Driven Development

In the application development phase, the framework user uses the metamodel and creates a model which solves a concrete problem. The creation of

concrete customization artifacts will be done by a generator based on the provided transformation rules. This only applies to these customizations which can be done declaratively. The imperative adaptations have to be done manually by the framework user.

3 Introspective Model-Driven Development

In this paper, we propose a bottom-up approach to realize model-driven development. We call this new approach *introspective model-driven development* (IMDD).

The main idea of IMDD is the construction of frameworks that can be analyzed in order to obtain the metamodel for customizations they define. The process in which the metamodel is retrieved is called *introspection*. The term introspection stems from the latin verb *introspicere*: to look within. Special emphasis should be put on the distinction between introspection and *reflection* in this context. We use both terms as they have been defined by the OMG [11]:

Table 1. Term Definitions

introspection	A style of programming in which a program is able to examine parts of its own definition. Contrast: reflection
reflection	A style of programming in which a program is able to alter its own execution model. A reflective program can create new classes and modify existing ones in its own execution. Examples of reflection technology are metaobject protocols and callable compilers.
reflective	Describes something that uses or supports reflection.

According to the definition of reflective, *introspective* describes something that supports introspection. An introspective framework supports introspection in that its metamodel can be examined.

The whole process of introspective model-driven development is schematically shown in figure 3. The process is divided into the well known core development phase and application development phase. The first result of the core development phase is an introspective framework. An introspective framework supports introspection by highlighting all declaratively customizable extension points through annotations [12]. This enables the extraction of the metamodel by *metamodel introspection*. It is important to understand, that the metamodel is not an artifact to be created from the framework developer, but rather can be retrieved at any point in time from the framework.

The central artifact of the application development phase are the customizations to be made by the framework user. In IMDD it is possible to analyze these artifacts and to obtain a model representation of them. This is called *model introspection*. The model is an instance of the retrieved metamodel and can be

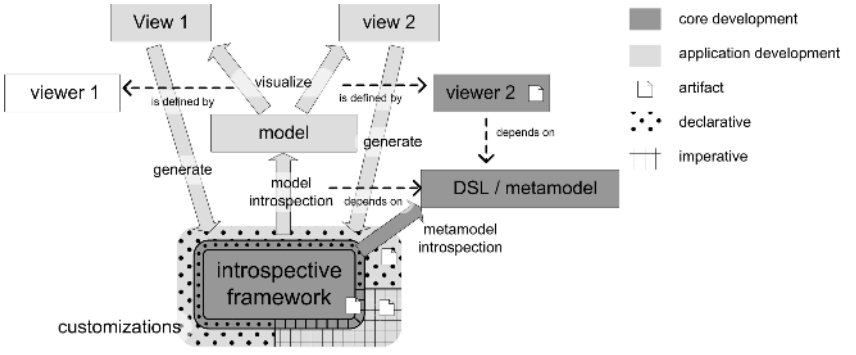

Fig. 3. Introspective Model-Driven Software Development

visualized by different viewers (i.e. visualization tools). There exist out-of-the-box viewers which can visualize an introspective model in a generic way. In some cases it is desirable to develop special viewers which visualize the model in a specific way. This will be done by framework developers in the core development phase. The manipulation of the model can be either done by using the views or by manipulating the customization artifacts directly. In both cases an updated customization artifact leads to an updated model and subsequently to an updated view. As a result of this, the model and the views are always synchronized with the actual implementation and can never "lie".

The main idea of introspective model-driven development is the direct extraction of the model and the metamodel from the framework artifacts which define them. There are two categories of frameworks which differ in the way adaptation takes place. *Blackbox frameworks* can be customized by changing association relationships flexibly. There are as many implementations as necessary to address all imaginable problems available as part of the framework core. The framework user just chooses the appropriate classes and configures their properties and the associations between them. In contrast, customization of *whitebox frameworks* takes place by creating subclasses of existing classes of the framework core. In this case the framework user has to provide concrete implementations.

Accordingly, the way introspective model-driven development is done is different for these kinds of frameworks. In the next chapter we will discuss introspective model-driven development for blackbox frameworks. In chapter 5, IMDD for whitebox frameworks will be introduced.

In order to enable introspective model-driven development we created a framework which supports blackbox introspection as well as whitebox introspection. This framework is called *Introspective Modeling Framework* – IMF. IMF provides its functionality by extending the post-IntelliJ-IDE Eclipse. Technically speaking, IMF consists of three Eclipse plugins.

Example. The process of developing an introspective framework and customizing it is illustrated using a simple example framework. We use a "textbook"

scenario described by Martin Fowler, in which we have a system that reads files and needs to create objects based on these files [2]. Each line can map to a different class, the class is indicated by a four-character code at the beginning of the line. The rest of the line contains the data for the object to be created. The following two lines result in the creation of two objects of type `ServiceCall` and `Usage` with attribute values as shown in an object diagram in figure 4:

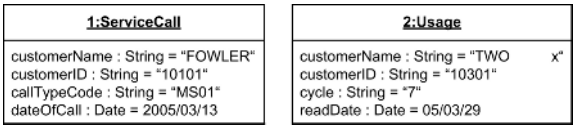

Fig. 4. Initialized Objects

```
#12345678901234567890123456789012345678901234567890123456789012345678901234567890
SVCLFOWLER 10101MS0120050313.........................
USGE10301TWO x50214..7050329.........................
```

The process of reading a file and instantiating objects accordingly should be adaptable in a high-level model-driven way. A conceptual metamodel which models the problem as an object-oriented design is shown in figure 5.

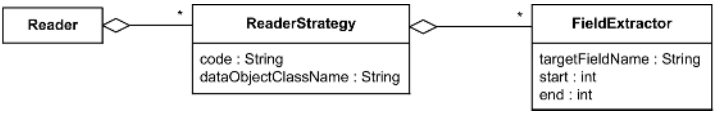

Fig. 5. Conceptual Metamodel

In the following we will show how to create an introspective blackbox and whitebox framework which realize a solution to this problem.

4 Blackbox Introspection

The framework core of a blackbox framework provides ready-to-use implementations of functionality, which only has to be customized to solve a family of related problems. The extension points of a blackbox framework are places which enable the adaptation of either elementary properties or associations between objects. In an introspective blackbox framework these extension points are tagged explicitly. That enables tool support for the customization process, which involves the selection and configuration of classes to be instantiated and the creation of associations between the objects constructed.

The idea of introspective blackbox frameworks is similar to that of *dependency injection* [13]. This means, that the instantiation of the framework classes is done by a dedicated component, which can be configured declaratively. The classes to be instantiated are rather passive in this process, they get their required dependencies "injected". Introspective blackbox frameworks take this idea one step further by declaring all resources to be injected explicitly. This enables tool support.

4.1 Core Development

The core of a blackbox framework consists of classes which can have configurable elementary properties and associations with other classes, which are also configurable. These configurable elements define the metamodel of the framework, and a concrete configuration is a model which has to conform to the metamodel.

The key point of *introspective* blackbox frameworks is that these configurable elements are tagged explicitly using annotations as being configurable. This enables the automatic introspection of the metamodel and as a result of this it is possible to support the modeling step.

There are two types of annotations, which enable the identification of configurable properties and associations. Configurable properties are tagged using the annotation type `Property`. In order to create the configurable property `code` of the class `ReaderStrategy` in our example, it is necessary to tag the definition of the attribute as shown:

```
@Property(description="these four letters indicate this strategy")
String code;
```

Configurable relationships are created using the annotation type `Association`. Creating the association between the classes `Reader` and `ReaderStrategy` is done with following piece of code:

```
@Association List<ReaderStrategy> readerStrategies;
```

This leads to the meta-metamodel of blackbox introspection as shown in figure 6. A configuration consists of many configurable classes which can have many customizable properties and associations. An association connects configurable classes with each other. The icons besides the classes `ConfigurableClass`, `Property` and `Association` can be used to annotate introspective elements in class diagrams.

An implementation of the example problem as an introspective blackbox framework looks like shown in figure 7. The introspective elements are annotated using the icons mentioned above.

4.2 Application Development

So far we have looked at how to create the introspective framework core. This task is done by the framework developer and consists in writing a "plain old

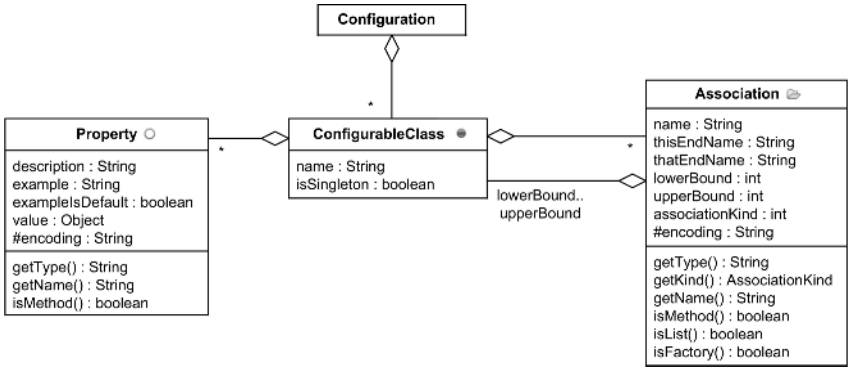

Fig. 6. The Meta-Metamodel of Blackbox Introspection

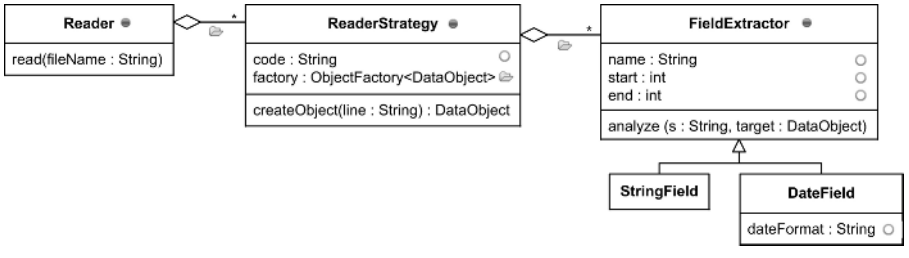

Fig. 7. Implementation of the Example Problem as an Introspective Blackbox Framework

framework" with some additional annotations to tag the extension points. As a result it is possible to retrieve the metamodel of the framework by doing introspection on the framework core.

In the second phase of the life cycle, the framework user customizes the framework to solve a concrete problem. The customization of an introspective blackbox framework is done using the *IMF Blackbox Modeler* tool. Technically speaking is this a plugin for the Eclipse IDE which analyzes the metamodel of the framework core. Based on this metamodel the Blackbox Modeler provides a view which enables the creation of a model which is an instance of the metamodel. A screenshot of the modeler, in which the example problem is modeled, is shown in figure 8. From the modeler view, which shows the model, it is always possible to navigate to the corresponding metamodel element, which is also shown in figure 8.

5 Whitebox Introspection

As already mentioned, the customization of whitebox frameworks is done by providing implementations of abstract classes of the framework core. More specifically, the framework user specifies the desired behavior by implementing methods.

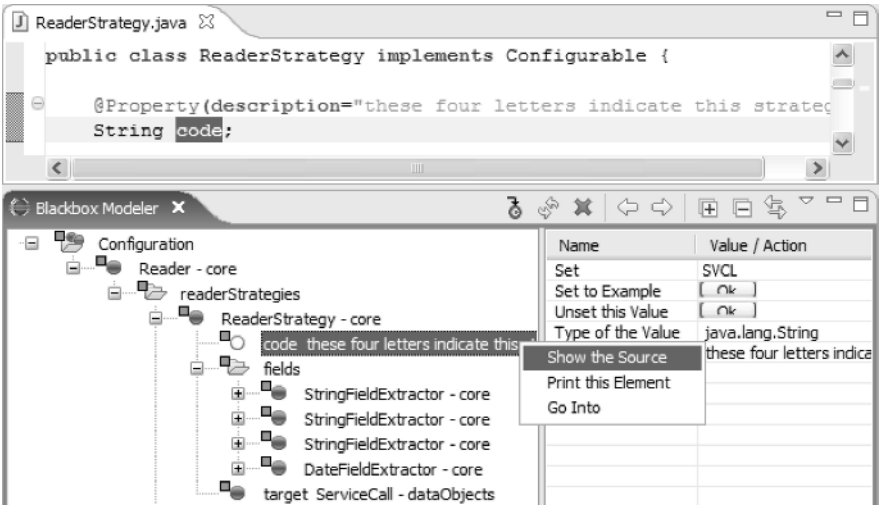

Fig. 8. Modeling Perspective for Blackbox Introspection in the Blackbox Modeler

These methods are called *hook methods* and represent the extension points of the framework [15]. Regarding introspective whitebox frameworks there are two kinds of hook methods – introspective and non-introspective hook methods. Customization of introspective hook methods can be done using a declarative programming style, while implementing non-introspective hook methods requires imperative constructs. The main idea of whitebox introspection is to annotate introspective hook methods in the framework core and to analyze the declarative customization artifacts. The analysis of the structure of the introspective methods results in the metamodel of the framework, and the analysis of the customizations leads to a model of the provided adaptations.

To build a whitebox framework, which addresses our example problem, we create an abstract class `ReaderStrategy`. This abstract class specifies, that subclasses have to provide a concrete value of the `code` property:

```
public abstract class ReaderStrategy {
    @Introspective public abstract String getCode();
    ...
```

The annotation type `Introspective` indicates, that this method is an introspective method, which means that it has to be implemented in a declarative way. In fact, this is the simplest kind of an introspective method, the so-called *value-method*. A value-method has no parameters and returns either a primitive value or an object of type `String` or `Class`.

In order to specify the programming model formally, which can be used to implement the method we use a context-free grammar. This grammar is used to restrict the expressive power of the underlying programming language to

a declarative programming model. We define our grammar based on the non-terminals used by the Eclipse project JDT [16]. Non-terminals are shown in italic type, terminal symbols are shown in fixed width font. Non-terminals introduced by us are printed in ***bold italic*** face.

The non-terminal which defines the programming model for customizing value-methods looks like the following:

ValueMethod_M1 :
{ Modifier } ***ValueMethodType*** SimpleName `(` `)` {
`return` ***AbstractValue*** `;` }

ValueMethodType :
`String` | `Class` | `boolean` | `byte` | `short` | `char` |
`int` | `long` | `float` | `double`

The return type of a value-method is therefore restricted to be of either primitive type or one of `String` or `Class`. The return statement is defined by the non-terminal ***AbstractValue***:

AbstractValue :
Value
$Name_{ValueVariableName}$

This can be either a value of one of the following types, or a variable name:

Value :
BooleanLiteral | *CharacterLiteral* | *NumberLiteral* |
StringLiteral | *TypeLiteral* | *NullLiteral*

We call these two ways to return the result *by-value* and *by-constant*. The variable name has to be bound to a field declaration which defines a variable which is declared as being final:

FinalValueFieldDeclaration :
[Javadoc] ***FinalModifiers***
ValueMethodType $SimpleName_{ValueVariableName}$ `=` ***Value*** `;`

The non-terminal ***FinalModifiers*** specifies a set of modifiers which contains the final modifier. A valid implementation of the introduced method **getCode** looks like the following:

```
@Override public String getCode() {
    return "SVCL";
}
```

An alternative solution in which case the result is returned by-constant is like the following:

```
final CODE = "SVCL";

@Override public String getCode() {
    return CODE;
}
```

Because of the declarative programming model, it is possible to analyze the customization artifacts. This analysis is called *model introspection* and is supported by the IMF-Whitebox Modeler tool. A screenshot, which shows the tool, is illustrated in figure 9. In this view, it is possible to manipulate the value of the property, which results in a manipulation of the code and a subsequent redrawing of the model. It is also possible to navigate to the construct which defines the metamodel for the current model element.

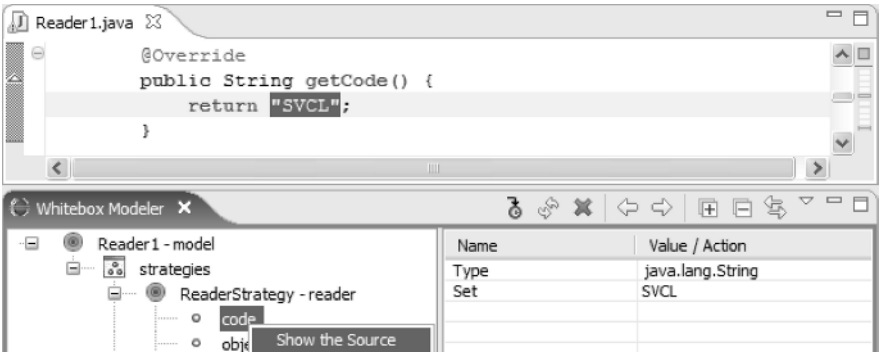

Fig. 9. An Introspective View of a Value-Method in the Whitebox Modeler

The value-method described so far enables us to model elementary properties. In order to build a framework which addresses the example problem we also have to model associations. This can be done using another kind of introspective method, the so-called *objects-method*. An objects-method has no parameters, but returns either one or many objects of a specific type. Unlike for the value-method, there are multiple introspective programming models, which can be used to implement an objects-method. The simplest programming model returns just a newly created object, which is similar to the programming model of the value-method. An in-depth treatment of all identified programming models will be provided in [14]. We introduce here the fields-by-type programming model, which allows the definition of a set of objects by declaring variables. The specification of an objects-method using the fields-by-type programming model in the framework core looks like the following:

```
public abstract class Reader {

    private List<ReaderStrategy> strategies;

    @Introspective public List<ReaderStrategy> getStrategies() {
        if(strategies == null) {
            strategies = FieldFinder.getFields
                (this,ReaderStrategy.class);
        }
        return strategies;
    }
    ...
```

This method returns all final fields which specify an object of type `ReaderStrategy`. A specification of a concrete strategy looks like this:

```
public class Reader1 extends Reader {
    final ReaderStrategy STRATEGY_1 = new ReaderStrategy() {
    ...
```

As a result of this we have introduced a meta-metamodel of whitebox introspection, which is shown in figure 10. The meta-metamodel we show here is a simplified version of the one introduced in [14]. A model consists of introspective classes. An introspective class has introspective methods, which can be either value-methods or objects-methods. Using this meta-metamodel it is possible to build introspective whitebox frameworks.

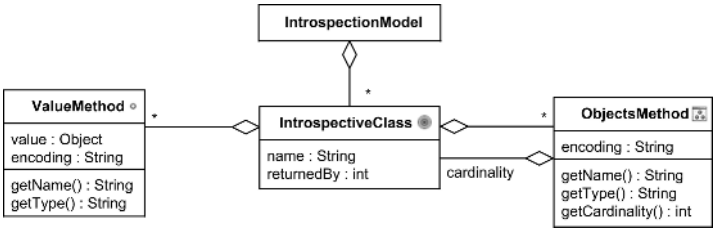

Fig. 10. The Simplified Meta-Metamodel of Whitebox Introspection

In figure 11 the four meta-layers and their equivalents in the case of whitebox introspection are shown. The metamodel at M2 is defined by the framework core by using introspective methods. The model at the meta-layer M1 is defined by the framework user. The model is represented as declarative code of the host language. Modeling can be done either by writing code manually or by using the Whitebox Modeler to do so at a rather high level of abstraction. The programming model, which is used to express the model, depends on the actual

introspective method. The Whitebox Modeler also verifies the correct use of the programming model.

We now study how to solve the example problem with an introspective whitebox framework. Our example can be customized completely declaratively, so it can be solved using a blackbox framework. To demonstrate one of the advantages of whitebox introspection we vary the example scenario a little bit. Let's assume, the created objects should be used to do some rather complex business logic directly after their creation. This business logic should be done using a specific API, which enables the manipulation of some data store. The most convenient way to express such kind of business logic is by writing some imperative code, which encodes the desired behavior. This means, that there are declaratively customizable parts of the framework as well as imperatively customizable part. One benefit of our approach lies in the uniform treatment of both introspective and non-introspective hook methods. The content of the introspective methods will be analyzed, whereas the non-introspective are not analyzed by the modeler.

A class diagram of an introspective whitebox framework, which addresses the modified example problem, is shown in figure 12. The method `processObject` of the class `ReaderStrategy` is a non-introspective hook method which gets the created object as a parameter and does the business logic. The framework user can use the full power of the base language to specify the business logic here.

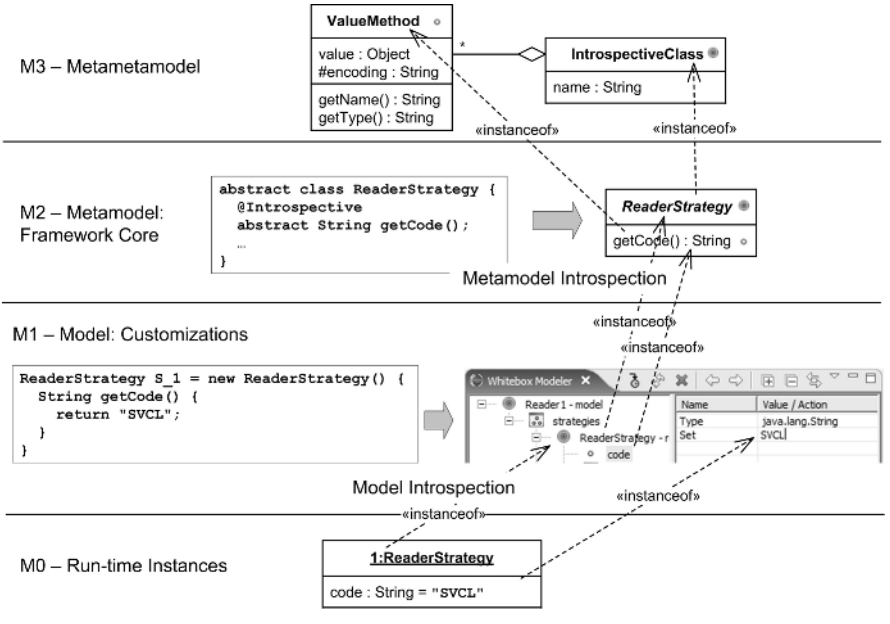

Fig. 11. Four Meta-Layers of Whitebox Introspection

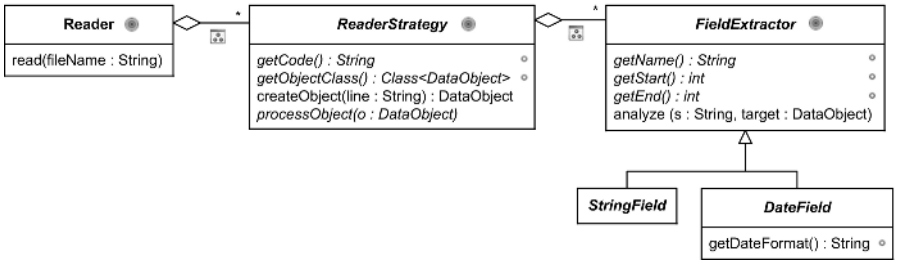

Fig. 12. Implementation of the Example Problem as an Introspective Whitebox Framework

5.1 Case Study

In [14] we present our experience in building two whitebox frameworks as part of a commercial knowledge management system. The first one is a web-visualization framework. The main abstraction of this framework are so-called *handlers*. A handler reacts on requests by reading parameters, doing some business logic and rendering response pages in the end. Except for the business logic, all aspects of the handlers are realized introspectively and can be analyzed and modeled. We have built a derivative of the Whitebox Modeler, which is tailored specifically to this framework. In order to render the dynamic response pages, the framework uses HTML-templates. By means of the introspective model, it is possible to check the consistency of the templates with the code which instruments them [17]. The whole knowledge management system consists of approximately 500 handlers.

6 Discussion and Concluding Remarks

We believe, that modeling as a means of building and understanding systems at a rather high level of abstraction should play a more important role in software engineering. Furthermore, we think, that model-driven approaches offer a lot of benefits over merely model-based approaches. The prevailing approach to realize model-driven development is the generation of artifacts which customize frameworks, as shown in figure 1. In this paper, we propose an alternative approach to realize MDD, which we call introspective model-driven development (see figure 13). In the following we will discuss the implications of using introspective vs. generative model-driven development.

IMDD relies on some infrastructure, which has to be in place. The base language used has to be a statically typed object-oriented programming language. In our case, we chose Java as the base language. The second prerequisite is the existence of a "post-IntelliJ-IDE", on top of which a tool to support IMDD can be created. We chose the Eclipse IDE to build IMF, which is a framework that supports IMDD. According to the two types of introspective frameworks - introspective blackbox and introspective whitebox frameworks, IMF provides generic

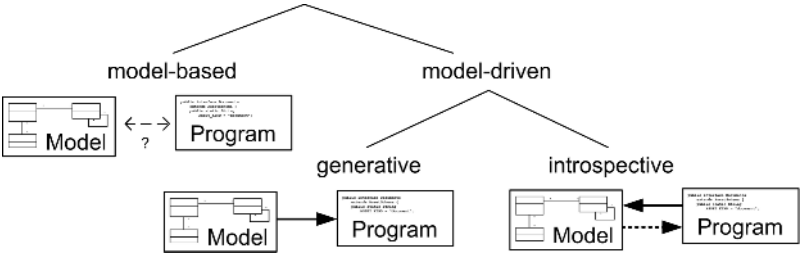

Fig. 13. Modeling Approaches

modelers for both of them. Therefore, using IMF out-of-the-box it is possible to do introspective model-driven development immediately. In some cases, it is useful to create tailored modelers, which will be supported by IMF.

6.1 Advantages of IMDD

IMDD is a *single-source* approach, which means that the metamodel and the model are respectively represented by exactly one artifact. This is not the case for generative approaches, in which information about the metamodel is encoded implicitly in the framework core, and in the explicit metamodel of the DSL. The same is true for the models. They are represented as artifacts of the modeling process, as well as customizations, which will be generated. To specify the transformation process, there are additional artifacts, which rely on the conceptual metamodel, and the way the concrete customization artifacts look like. All this leads to a lot of redundancies and a lot of artifacts, which have to be consistent.

In introspective model-driven development the metamodel is represented by the annotated framework core, and the model is represented directly by the customization artifacts. In both cases, these artifacts are used to specify the executable system, as well as to provide modeling information. This means, that in IMDD "code is model" [18]. Code means here also declarative customization artifacts, which configure an introspective blackbox framework. The model is a transient view on the underlying code. The most striking advantages of IMDD follow from this fact.

At first, this enables roundtrip visualizations, which are hard to achieve for generative approaches [5]. As another immediate implication of this, the model "never lies". This means, that the model reflects properties of the system precisely all the time.

Because the modeling information in IMDD is represented by code, refactoring the metamodel of the framework [19] can be done easily using a post-IntelliJ-IDE. In the case of an introspective whitebox framework, also the model will be refactored accordingly. Broadly speaking, keeping the involved artifacts consistent is quite easy in IMDD. In a generative approach, evolving the framework core means evolving the metamodel, the transformation rules and the models manually in parallel.

Another advantage of IMDD is the possibility to achieve symbolic integration [2] between declarative models and imperative artifacts. This makes it easy to mix both programming styles, and get the benefits of modeling the declarative aspects on a high level of abstraction. Using a generative approach, it is quite complicated to integrate both paradigms, by e.g. editing generated artifacts, using protected source code areas.

Furthermore, we consider the introspective approach as being *lightweight*. This means, that no additional meta-metamodel is needed to do IMDD, and that the overall process is much simpler. As a meta-metamodel, we use some of the capabilities of the object-oriented base language. There are no additional languages to be learned by the developer. Generative approaches are more heavyweight, because they involve an additional meta-metamodel, and a language to do the transformation. As another aspect of using the base language to do metamodeling, fundamental consistency constraints on the metamodel will be checked by the compiler of the base language. In the case of introspective whitebox frameworks, the compiler also checks some aspects of the well-formedness of the model using rich typing, binding and scoping rules of the base language.

The code-centricity of IMDD matches well with the development approaches used in practice.

6.2 Disadvantages of IMDD

IMDD relies on the explicit annotation of the extension points in the framework core, so it requires the construction of introspective frameworks. It is not possible to do IMDD with classical frameworks, which do not support this development approach. As a consequence of this, doing introspective development with the existing frameworks is not possible. They have to be modified, in order to be introspective.

References

1. Ralph E. Johnson and Brian Foote, *Designing reusable classes.* Journal of Object-oriented Programming, vol. 1(2), pp. 22-35, 1988.
2. Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?.* http://www.martinfowler.com/articles/languageWorkbench.html
3. Markus Völter and Thomas Stahl, *Model-Driven Software Development.* John Wiley & Sons, 2006.
4. Elliot J. Chikofsky and James H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, vol. 7, 1990.
5. Shane Sendall and Jochen Küster, *Taming Model Round-Trip Engineering.* Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Vancouver, Canada, 2004.
6. Stuart M. Charters, Nigel Thomas, and Malcolm Munro, *The end of the line for Software Visualization?.* VISSOFT 2003: 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis, Amsterdam, September 2003.

7. David S. Frankel, *Model Driven Architecture – Applying MDA to Enterprise Computing*. Wiley Publishing, Inc., 2003.
8. Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, *Software Factories*. Wiley Publishing, Inc., 2004.
9. Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose, *Eclipse Modelling Framework*. Addison-Wesley Professional, 2003.
10. OMG – Object Management Group, *Meta Object Facility (MOF) 2.0 Core Specification*. http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-15.pdf
11. OMG – Object Management Group, *Common Warehouse Metamodel (CWM), v1.1 – Glossary*. http://www.omg.org/docs/formal/03-03-44.pdf
12. Joshua Bloch, *JSR 175: A Metadata Facility for the Java Programming Language*. http://www.jcp.org/en/jsr/detail?id=175
13. Martin Fowler, *Inversion of Control Containers and the Dependency Injection Pattern*. http://www.martinfowler.com/articles/injection.html
14. Thomas Büchner, *Introspektive modellgetriebene Softwareentwicklung*. Technische Universität München, München, Dissertation (in Vorbereitung).
15. Wolfgang Pree, *Essential Framework Design Patterns*. Object Magazine, vol. 7, pp. 34-37, 1997.
16. Eclipse Foundation, *Eclipse Java Development Tools (JDT) Subproject*. http://www.eclipse.org/jdt/
17. Stefan Käck, *Introspektive Techniken zur Sicherung der Konsistenz zwischen Webpräsentationsvorlagen und Anwendungsdiensten*. Diplomarbeit, Technische Universität München, 2005.
18. Harry Pierson, *Code is Model*. https://blogs.msdn.com/devhawk/archive/2005/-10/05/477529.aspx
19. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.