Magritte – A Meta-Driven Approach to Empower Developers and End Users[⋆]

Lukas Renggli[1], Stéphane Ducasse[2], and Adrian Kuhn[1]

[1] Software Composition Group, University of Bern, Switzerland
{renggli,akuhn}@iam.unibe.ch
[2] LISTIC, University of Savoie & INRIA Futurs Lille, France
stephane.ducasse@free.fr

Abstract. Model-driven engineering is a powerful approach to build large-scale applications. However, an application's metamodel often remains static after the initial development phase and cannot be changed unless a new development effort occurs. Yet, end users often need to rapidly adapt their applications to new needs. In many cases, end users would know how to make the required adaptations, if only the application would let them do so. In this paper we present how we built a runtime-dynamic meta-environment into Smalltalk's reflective language model. Our solution offers the best of both worlds: developers can develop their applications using the same tools they are used to and gain the power of meta-programming. We show in particular that our approach is suitable to support end user customization without writing new code: the adaptive model of Magritte not only describes existing classes, but also lets end users build their own metamodels on the fly.

Keywords: Meta-Modeling, Meta-Data, Adaptive Object Model, Business Application Development, Smalltalk

1 Introduction

As a result of our experience with developing dynamic web applications at an industrial scale[3], we recognized the need to introduce a meta-layer to provide us with more flexibility. Describing domain entities is not a new idea [1–5]. However, often meta-descriptions remain static after the initial development phase and cannot be changed unless a new development effort occurs. Yet, end users often need to rapidly adapt their applications to new business needs [6] and in many cases, they would know how to make the required adaptations, if only the application would let them do so [7].

Application requirements usually do not remain static after the initial development phase. Changing business plans typically boils down to minor modifications to domain objects and behavior, for example new input fields have to

[⋆] In *Model Driven Engineering Languages and Systems*, Gregor Engels, Bill Opdyke, Douglas C. Schmidt and Frank Weil (Eds.), LNCS, vol. 4735, Springer-Verlag, September 2007, pp. 106-120.

be added, configured differently, rearranged or removed. Unfortunately most of today's applications don't provide this ability to their end users. The situation is even more striking in the context of web applications that are typically built for a lot of different people with varying needs. Furthermore it is often the case that software systems have a static object model: one that has been defined by the software architect at implementation time and that cannot be changed later without changing and recompiling the source-code.

Generative techniques should be avoided, as they prevent the metamodel from being dynamically changed at runtime. Also, the introduction of meta-descriptions should not disrupt the normal way of programming and the tools used to program. The development tools (refactorings, version control, unit testing, debugger, etc.) should continue to work as if there were no meta-descriptions [9]. The approach should be integrated as closely as possible into the object-oriented paradigm, the tools and the programming environment. In our case we use Squeak[4], an open-source Smalltalk [10, 11], and Seaside[5], an open-source web application framework [12].

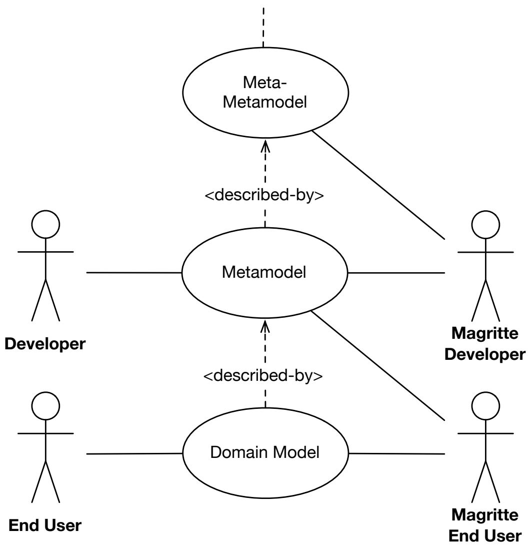


Fig. 1. Magritte is self-described and features metamodel changes at runtime. This allows end users not only to interact with the application data, but also change the metamodel without having to write code.

[3] The first author of this paper is an independent consultant and software architect. In the context of his Master thesis [8] he invented and developed the Magritte framework, which is used in several large-scale industry and open-source projects.

[4] http://www.squeak.org

[5] http://www.seaside.st

This publication reports on our experience with using the Magritte meta-descriptions framework. Magritte has been originally developed for web applications, but its applicability goes beyond that context. The Magritte meta-descriptions are integrated into the reflective metamodel of Smalltalk to support the development of flexible applications. As the Magritte metamodel is self-described, it is possible to apply the same editors for both domain data and its corresponding metamodel. As illustrated on Figure 1 this enables a Magritte user to work on two meta-levels at the same time. This applies to both the end user and the developer. With Magritte we can reap the benefit of the two worlds: On the one hand we keep our efficient and dynamic object-oriented programming with an excellent tooling context, and at the same time we gain the flexibility and compactness of meta-descriptions to factor repetitive tasks of our application development.

This paper is structured as follows: Section 2 introduces the Magritte framework and presents an example how Magritte descriptions are specified. In Section 3 we present different interpreters that have been written for Magritte. Section 4 explains how Magritte is self-described and how this enables end users to customize their applications. Section 5 compares Magritte to related frameworks and Section 6 evaluates our approach and discusses the lessons learnt.

2 Describing Domain Objects

Magritte is a meta-description framework, describing domain classes and their respective attributes, relationships and constraints [4]. Magritte augments the reflective metamodel of Smalltalk [13] with additional means to reason about the structure and behavior of objects. The Smalltalk programming language is used to define Magritte meta-entities and their behavior. An attribute description contains the type information, the way the attribute is accessed, and some optional information such as a comment and label, relationships and validation conditions.

In the following sections we use the example of a meta-described person domain-model. The Person class defines the instance variables name and birthday. In Sections 4 and 6 we present more realistic examples used in productive applications.

Example. To describe the entities in this model we need corresponding description instances, that can be either built from the source-code at development time, dynamically at run-time, or a combination of the two approaches. Either way, the code to build the descriptions looks the same. To describe the name, we create an instance of StringDescription, define an access strategy (in this case the getter method #name is used), provide a label and add the constraint that this is a required value[6].

```
(StringDescription new)
    selectorAccessor: #name;
    label: 'Name';
    beRequired
```

Note that descriptions provide much more information than just type information. A date description, for example, knows how the attribute should be displayed (June 11, 1980, 11 June 1980, 06/11/1980), edited (text-input fields, drop-down boxed, date-picker), and validated. Moreover descriptions do not necessarily describe instance variable attributes, but might also describe derived attributes that are dynamically calculated on demand.

2.1 Structural Descriptions

The Essential Meta-Object Facility (EMOF) is a standard for model driven engineering defined by the Object Management Group (OMG). Similar to EMOF Magritte is not designed as a layered architecture. Magritte descriptions live in a flat world and there is no distinction drawn between objects in the meta-metamodel (M3), the metamodel (M2), the model (M1) and the instances (M0).

Contrary to EMOF Magritte has no notion of instantiation, inheritance and classes. We describe objects that have already been instantiated. Magritte is tightly embedded into the Smalltalk object model. Smalltalk is used to instantiate, configure and compose the descriptions, as well as to model the behavior of the meta-descriptions. In Magritte objects are not tightly connected with a single description. Descriptions can be shared, exchanged and applied to different instances and classes.

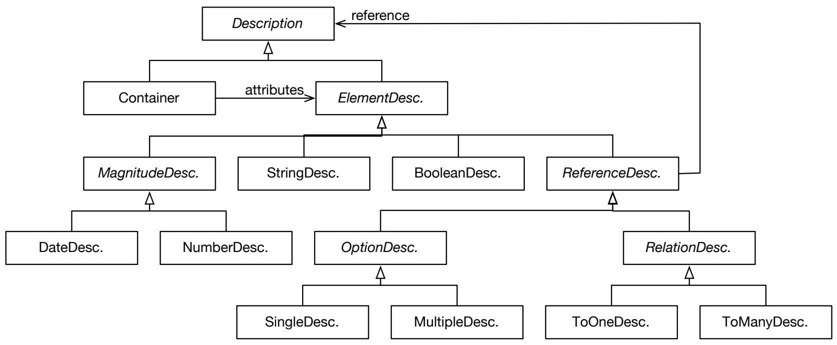


Fig. 2. The Description Hierarchy of Magritte

As seen in Figure 2 the description classes define a type hierarchy. This is similar to the subclasses of Type in EMOF, where a distinction between classes

[6] In Smalltalk messages follow the pattern receiver methodName: argument, which is equivalent to the Java syntax receiver.methodName(argument). Hence StringDescription new sends the message new to the class StringDescription that returns a new instance of the receiving class. Subsequently the messages selectorAccessor:, label: and beRequired are sent to this instance.

and primitive types is made. An instantiated Magritte description is similar to an EMOF property.

Magritte defines multiplicities using the Composite design pattern. The class ReferenceDescription knows another description, that is used to describe the referenced object. Whether the elements are *ordered* and/or *unique* is determined as a property in ReferenceDescription. Upper and lower bounds of are specified using constraints. In EMOF multiplicities are part of the type information. Our approach has shown to be more straightforward when automatically building editors and reports.

Option Descriptions. The SingleOptionDescription models an $1:1$ relationship. The class MultipleOptionDescription models a $1:n$ relationship. In both cases the referenced objects must be chosen from a list of existing objects satisfying the reference description.

Relationship Descriptions. The ToOneRelationshipDescription models an $1:1$ relationship. The ToManyRelationshipDescription models an $1:*$ relationship. In both cases any object can be referenced that satisfies the reference description.

The architecture of Magritte, *i.e.,* describing Smalltalk class with descriptions, is not new and can be seen as a validation of the nowadays well-known distinction between two conceptually different kinds of instance-of relationships: (1) a traditional and implementation driven one where an instance is an instance of its class, and (2) a representation one where an instance is described by another entity [14]. Atkinson and Kühne named these two forms: form vs. contents or linguistic and logical [15, 16].

2.2 Executability and Constraints

Magritte does not provide specific functionality to describe behavioral aspects, such as operations, their parameters and return values [17, 9]. This is not necessary, as methods in Smalltalk are objects that can be described as any other object. Then using the reflective facilities it is possible to retrieve a list of invokable method sends (first class method invocations) that are available on a particular class. On request these methods can be invoked with arguments provided by end users. This shows how Magritte integrates with the reflective facilities of Smalltalk. Furthermore Magritte directly supports constraint objects on its descriptions, that are similar to the constraints part of the Complete Meta-Object Facility (CMOF). We avoided introducing a specific constraint language, such as OCL, but use plain Smalltalk expressions. This simplifies the development, as developers can use the well known tools and don't have to learn a new language. As OCL was influenced by Smalltalk, our constraint expressions resemble those of OCL.

Example. To add a size constraint to a string description we use a block closure (anonymous function) to ensure a maximal size of 5 characters. In case the condition is not satisfied the error message "too long" is displayed:

```
aDescription addCondition: [ :value | value size <= 5 ] label: 'too long'
```

3 Interpreting Descriptions

Magritte descriptions can be interpreted in many different ways. Simple interpreters just iterate over the descriptions and perform different tasks on the associated model. In more generic cases we exploit the Visitor design pattern to walk trough the description graph. The most immediate use case is the one to automatically build views, editors and reports.

3.1 Building a View

The simplest interpreter that can be written is one that iterates over all descriptions of a domain model and prints the label and the current values onto a text stream. The following code shows everything that is needed to accomplish this task on any described domain-model as aModel in the following:

```
aModel description do: [ :desc |
    aStream
        nextPutAll: (desc label);
        nextPutAll: ': ';
        nextPutAll: (desc toString: (desc accessor readFrom: aModel));
        cr ].
```

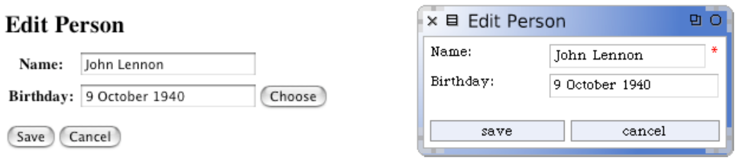
First we ask the model for its description, then we iterate over its individual attributes. Within the loop, we first print the label, then we ask the accessor of the description to return the associated attributes from aModel and transform this value to a string, so that it can be appended to the output. The resulting output might look like:

```
Name: John Lennon
Birthday: 9 October 1940
```

Since every description knows how to convert its values to strings, we get a readable list of all the described attributes of our domain-model. By defining a different string-conversion strategy in descriptions, we are able to change the way values are printed. When adding, removing or changing descriptions in the domain-model, the above code will still print the correct output without having to change a single line of the interpretation code.

3.2 Building an Editor

Most business applications today consist of a large number of input-dialogs that need to be built and validated manually. One of the goals of Magritte was that developers could specify how their domain objects can be modified, so that it becomes possible to automatically build editors for different user-interfaces frameworks, as seen in Figure 3.


Edit Person

Name: John Lennon

Birthday: 9 October 1940

Edit Person

Name: John Lennon

Birthday: 9 October 1940

Fig. 3. Interpreting descriptions for different GUI frameworks: the web (left) and Morphic Squeak (right)

Sending the message `asComponent` to a domain model returns a ready-to-use Seaside component that can be plugged into the web application. As in Section 3.1, Magritte will iterate over the descriptions and compose an editor. The default interpreter creates XHTML markup that is annotated with a variety of CSS classes, so that the layout and look can adapted to most needs by only using a different style-sheet. For specific cases it is always possible to subclass the interpreter or to define a different XHTML generation strategy on a per meta-description bases.

During an edit operation, Magritte works on copies of the values being edited, so that the original data remains untouched. Before actually committing the changes, Magritte checks if the model satisfies all its validation conditions. Moreover the framework ensures that there are no edit conflicts caused by other people editing the same objects at the same time, and, if necessary, shows a warning.

All this is very convenient for software developers, as they don't have to do the caching, the validation and the conflict detection for every editor manually. Not only does this increase the development speed, but it also makes the software more robust, since all editing concerns are handled at a single place and are not spread across all editors in the system.

3.3 Other Interpreters

Over the past few years many Magritte interpreters have been written:

Validate, Verify and Setup Objects. Whenever user input is requested incoming data has to be validated. Existing graphs of objects need to be verified from time to time to ensure validity. Complex graphs of objects need to be built and initialized with default values. All these tasks can be accomplished by walking trough a description graph and validate or build these objects on the fly.

Persistency, Indexing and Querying. Making objects persist is one of the most daunting tasks. Magritte description are able to tell a interpreter how a graph of domain models should be stored and loaded. For example Magritte can generate SQL statements to retrieve and update objects in a relational database. In the context of object databases it is crucial to build indexes to be able to efficiently query that data. With Magritte these tasks can be automated.

Introspection, Reflection. The metamodel of Magritte provides additional information that can be used to improve the development processes, for example in the debugger and in the inspector a high level view can be provided instead of a straight memory dump of the object layout.

4 End Users Customizability

Often dialogs in applications remain static after the initial development phase and cannot be changed unless a new development effort occurs. Yet end users often need to rapidly adapt their applications to new business needs. In many cases they would know how to make the required adaptations, if only the application would let them do so.

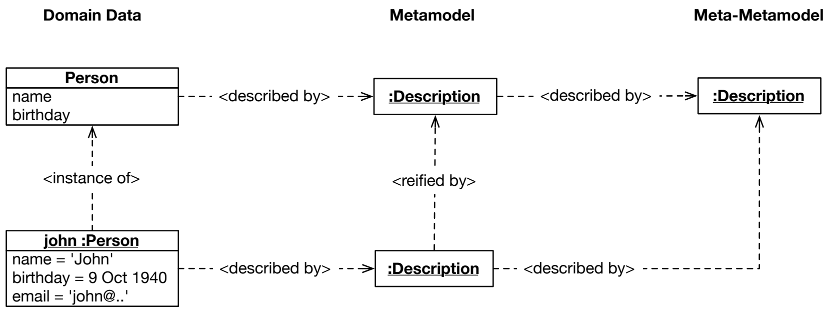
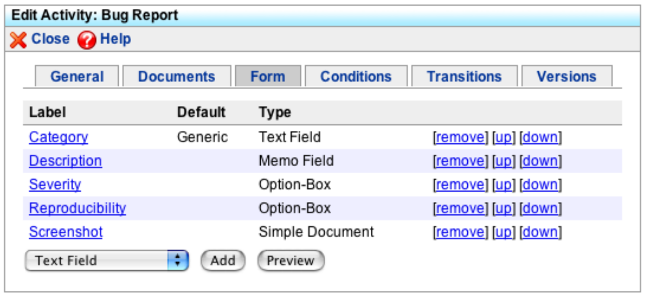


Fig. 4. The meta-levels of the Magritte metamodel.

As shown in Figure 4 there are different spots to specify, reify and interpret the Magritte metamodels. The domain class Person is written by the application developer. The class is described by a set of Magritte descriptions that are common to all instances of Person. These descriptions are either hardcoded into the source-code of the application or have been specified at runtime by an end user.

john is an instance of the class Person. The instance itself references a set of instance specific descriptions used to reify the class-based descriptions. These descriptions are either dynamically built from the application logic or have been manually specified by an end user using an editor as seen in Figure 5. Instances that do not use instance-specific descriptions simply reference the set of class descriptions. Furthermore to avoid the need to introduce an instance variable to hold the instance-specific descriptions on all objects, we propose the use of an adaptive model as presented in the next section.

Figure 5 shows a description editor that is part of a commercial workflow definition and runtime engine. The editor opened on a specific workflow task allows end users to customize the existing metamodel to suit their particular needs. Moreover, the end user is able to specify validation and transition conditions in

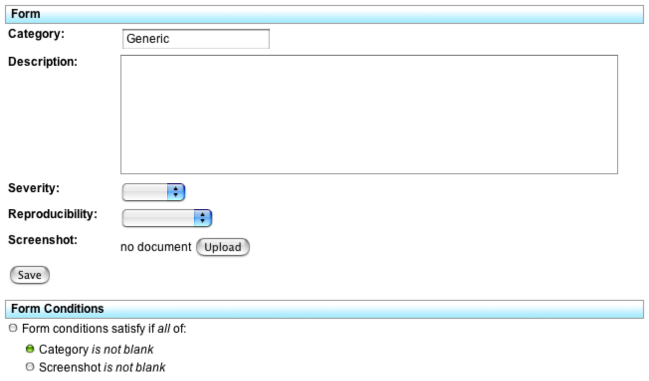


Edit Activity: Bug Report

Label	Default	Type	
Category	Generic	Text Field	[remove] [up] [down]
Description		Memo Field	[remove] [up] [down]
Severity		Option-Box	[remove] [up] [down]
Reproducibility		Option-Box	[remove] [up] [down]
Screenshot		Simple Document	[remove] [up] [down]

Fig. 5. A Magritte description editor that allows end users to change the metamodel without writing code.

different sections of the user interface. When interpreted by the runtime engine, see Figure 6, the specified metamodel is used to collect the data from the users and to operate the workflow execution. Therefore, an end user can adapt forms on the fly and see its effects directly. Furthermore the customized metamodel is exploited to operate reporting and querying facilities on running workflows. It is Magritte too, that is responsible to make all the meta-data and data persistent.



Form

Category:	Generic
Description:	
Severity:	
Reproducibility:	
Screenshot:	no document

Form Conditions

Form conditions satisfy if *all* of:

- Category *is not blank*
- Screenshot *is not blank*

Fig. 6. An automatically built editor from runtime customized meta-descriptions displaying if the form conditions are satisfied.

As we have illustrated in Figure 1, the possibility to work on two meta-level applies to both end users and developers. This, and the fact that Magritte describes itself, are the key concepts to enable end users to modify the metamodel on their own.

4.1 Adaptive Model: Enabling end user editable Meta-Descriptions

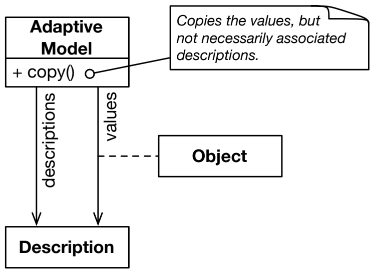

Adaptive Model
+ copy()
Copies the values, but not necessarily associated descriptions.
descriptions
values
Object
Description

Fig. 7. An adaptive model, mapping a set of descriptions to actual model values.

To enable instance specific metamodels, Magritte introduces a generic object model mapping descriptions to actual values, as seen in Figure 7. The AdaptiveModel has two instance variables, the first being used to refer to the descriptions of the instance and the other one to keep a list of the actual values of the model. Transforming the class of the adaptive model into a Trait [18] allows us turn any existing class into an adaptive model and to combine the descriptions defined in the class with the ones provided by the instance.

End users are able to edit the adaptive model at two different levels, at the model and at the metamodel level:

Domain Data Editing. Since the adaptive model is described, an editor can be built automatically (see Figure 6). The only difference is that the described values are not stored in instance variables of the model, but are kept within a hash table inside the adaptive model, mapping descriptions to their actual values. This gives much better flexibility when descriptions are added and removed.

Metamodel Editing. The descriptions of an adaptive model can change on the fly, since they are stored as part of the model-data. The descriptions can be either changed programmatically by the developer, or through end user interactions from a description editor. Since descriptions are described as well (see Figure 4), it is possible to let Magritte build a meta-editor (see Figure 5).

Descriptions can be shared among different adaptive model instances or can be unique to every instance. Therefore when copying an adaptive model one has to specify if the descriptions should be copied as well. If descriptions are shared, editing the metamodel affects all its associated instances.

5 Related Work

Yoder *et al* propose the type-square design pattern [19], based on the type object that separates the entity from its entity type [20]. Magritte uses these patterns as well, but it makes some generalizations, as seen in Figure 8: the distinction between components and properties is not made. A component and a property are just any kind of object. It is the same for component-types and property-types. They are all descriptions with the same superclass.

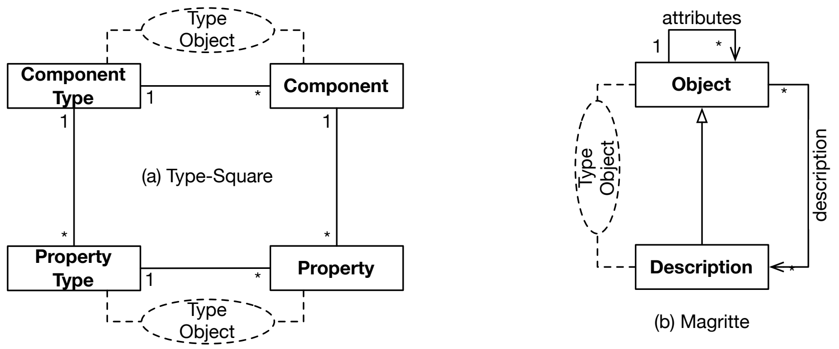


Fig. 8. (a) The type-square, and (b) the meta-recursive model of Magritte are both making extensive use of the type-object design pattern.

JavaBeans [5] includes a property framework similar to the description hierarchy of Magritte. However JavaBeans properties are solely based on the static type signature of the instance variable. Other settings, such as if the value is read-only, is determined implicitly through the absence of a write-accessor. JavaBeans properties do not describe themselves.

One reason that most frameworks do not describe themselves is that they all tend to be very domain-specific: some concentrate on the modeling of a specific business model, others concentrate on a specific output format, such as for a web framework. Unfortunately this leads to a model that is not able to describe itself. Therefore a lot of additional work is required if end users should be able to modify the adaptive-models. Magritte tries to consolidate everything by enabling meta-editing using itself.

Muller et al [21] present an approach to platform-independent web application modeling and development in the context of model-driven engineering. A specific metamodel (and associated notation) is introduced and motivated for the modeling of dynamic web specific concerns. Web applications are represented via three independent but related models (business, hypertext and presentation). A kind of action language (based on OCL and Java) is used on these models to write methods and actions, specify constraints and express conditions.

WebML [22] enables the high-level description of a web site according to distinct orthogonal dimensions: its data content (structural model), the pages that compose it (composition model), the topology of links between pages (navigation model), the layout and graphic requirements for page rendering (presentation model), and the customization features for one-to-one content delivery (personalization model). WebML goes in the same direction as Netsilon: An application is modeled using different perspectives and generated. Our approach is different. Our object-oriented applications are implemented in Smalltalk but meta-described, and this connected meta-description is used to support the generation of web user interface, queries and persistency. There is no automatic code generation involved in our approach, therefore if the metamodel changes, all the users of the metamodel behave the new way automatically.

6 Evaluation and Lessons Learned

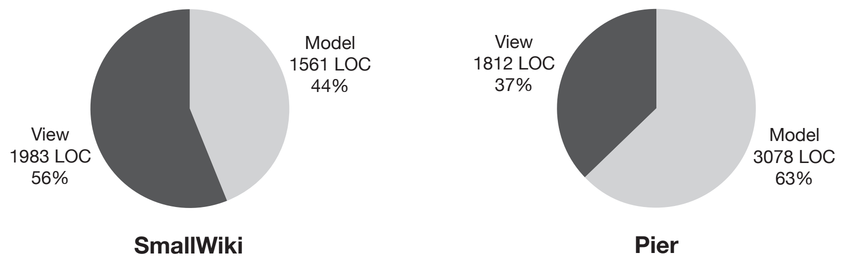


Fig. 9. Comparison of two web applications: SmallWiki and Pier (meta-described).

Figure 9 compares two web-based content management systems, SmallWiki[7] and Pier[8]. Both open-source systems have been written by the first author and make it possible to collaboratively build web sites. SmallWiki does not have a metamodel, all its features, such as the different views, the search engine, and the persistency, are hardcoded. Conversely Pier, the successor version of SmallWiki, is built from ground up using the Magritte metamodel. While the code base of Pier is noticeably bigger, it also provides functionality that was not possible in SmallWiki:

– As opposed to SmallWiki, Pier has a low coupling between model and view. Different views are interchangeable and their implementations are relatively small, as they only consist of Magritte glue code and some view specific functionality.
– Pier is easily extensible as the entities of the model are specified declaratively. The search engine, the persistency layer and user interface builders all take advantage of the Magritte descriptions.
– Most aspects of Pier can be customized by end users at runtime without having to write code. Additional data fields can be added to any page, to make it simple to collect and display structured data on the web.

[7] http://smallwiki.unibe.ch/smallwiki
[8] http://www.lukas-renggli.ch/smalltalk/pier

As stated by Ralph Johnson [6] a metamodel introduces additional complexity to an application and therefore inexperienced developers might have conceptual problems. Another problem might be a reduced execution speed, as there are additional indirections introduced through the interpretation of the metamodel. Comparing the execution speed of two systems like SmallWiki and Pier is difficult, as their features and implementation details don't exactly match. Most of the time other factors such as the network connection and persistency back-ends are more critical than the use of an underlying metamodel.

To evaluate the speed penalty when using a metamodel we benchmarked the text search of the two frameworks. Both frameworks are using the Visitor pattern to walk over the object graph: in SmallWiki this is hard coded, while in Pier this makes extensive use of meta-descriptions. For the benchmark we created a test setup of 100 pages and run 100 queries on both implementation. As expected SmallWiki performed better with a cumulated search time of 2456 ms. In Pier the search took 8190 ms, so the meta-driven search is about 30% slower than the hard coded one. Given the number of involved objects (a single page consists of hundreds of described objects) text search is a hard task for the meta-driven approach, as many descriptions have to be traversed and matched on the fly. We expect a much better performance for other use-cases and we plan to perform compile-time caching if necessary.

We have described our experience of using a metamodel integrated in the reflective metamodel of Smalltalk to support the development of flexible application. Our metamodel is self-described which enables end user customization.

6.1 Lessons Learned

As we have observed while developing several real world applications, having a meta-framework such as Magritte greatly reduces recurrent work, such as implementing different views, editors and persistency. Often it is much simpler to write a generic interpreter of the metamodel than to manually build specific implementations of the functionality in different places of the application. Developers only change the description at one single place in the source-code without having to refactor all places that deal with the object. More important end users are enabled to reify the choices of the developer through a convenient interface without having to know anything about the implementation and the underlying programming language. Hence, the use of Magritte not only supports the developers, but it makes the application more adaptable to changing needs of end users and reduces the need for development iterations.

Extending the existing Smalltalk metaclass does not allow to keep the metamodel independent of the actual implementation of the class. It should be possible to exchange the metamodel on the fly, and even use multiple metamodels at the same time for the same underlying domain object. Moreover we would like to let end users customize these model, without that they have to know the underlying programming language.

The fact that descriptions are used to describe Magritte itself, makes the system even more versatile: it gives end users the possibility to customize existing models or to build new ones, without having to write a single line of code. The interpreting software system can easily control how far this meta-customization should go. We observed that exposing a small subset of Magritte to end users greatly reduces complexity and increases productivity. Having adaptive models is the key for customizable applications, to allow end users build their own data-models.

As future work, we would like to investigate how the control flow of applications could be meta-described with Magritte. Especially in the context of web application it would be interesting to model the flow between pages as a meta-described graph, and again end user should be empowered to customize it on the fly.

Acknowledgments

We thank Oscar Nierstrasz for his careful review of a draft of this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, Capturing and Taming Software Change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872).

References

1. Group, O.M.: Common warehouse metamodel. Technical report, Object Management Group (2003)
2. Group, O.M.: Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group (2004)
3. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley Professional (2003)
4. Union, I.T.: Abstract syntax notation one (ASN.1). Technical report, International Telecommunication Union (2002)
5. Hamilton, G.: Javabeans. Technical report, Sun Microsystems (1997)
6. Yoder, J.W., Johnson, R.: The adaptive object model architectural style. In: Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02). (2002)
7. Atkinson, B.: Hypercard. Hypercard (1987)
8. Renggli, L.: Magritte – meta-described web application development. Master's thesis, University of Bern (2006)
9. Ducasse, S., Gîrba, T.: Using Smalltalk as a reflective executable meta-language. In: International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006). Volume 4199 of LNCS., Berlin, Germany, Springer-Verlag (2006) 604–618
10. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (1983)
11. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press (1997) 318–326

12. Ducasse, S., Lienhard, A., Renggli, L.: Seaside — a multiple control flow web application framework. In: Proceedings of 12th International Smalltalk Conference (ISC'04). (2004) 231–257
13. Rivard, F.: Smalltalk: a reflective language. In: Proceedings of REFLECTION '96. (1996) 21–38
14. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proceedings Automated Software Engineering (ASE 2001), Los Alamitos CA, IEEE Computer Society (2001) 273–282
15. Atkinson, C., Kuehne, T.: Concepts for comparing modeling tool architecture. In: Proceedings of the UML Conference. Number 3713 in LNCS (2005) 19–33
16. Atkinson, C., Kuehne, T.: The essence of multilevel metamodeling. In: Proceedings of the UML Conference. Number 2185 in LNCS (2001) 19–33
17. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In L. Briand, S.K., ed.: Proceedings of MODELS/UML'2005. Volume 3713 of LNCS., Montego Bay, Jamaica, Springer (2005) 264–278
18. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland (2002) Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.
19. Yoder, J., Balaguer, F., Johnson, R.: Architecture and design of adaptive object models. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). (2001) 50–60
20. Johnson, R., Wolf, B.: Type object. In Martin, R.C., Riehle, D., Buschmann, F., eds.: Pattern Languages of Program Design 3. Addison Wesley (1998) ISBN:0-201-31011-2.
21. Muller, P.A., Studer, P., Fondement, F., Bézivin, J.: Independent web application modeling and development with netsilon. Software and System Modeling **4** (2005) 424–442
22. Ceri, S., Fraternali, P., Bongio, A.: Web modeling language (WebML): a modeling language for designing web sites. In: Ninth International World Wide Web Conference. (2000)