

Magritte

Meta-Described Web Application Development

Masterarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Lukas Renggli

Juni 2006

Leiter der Arbeit

Prof. Dr. Stéphane Ducasse

Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Further information about this work, the tools used and an online version of this document can be found at the following places.

Lukas Renggli
renggli@gmail.com
<http://www.lukas-renggli.ch>

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
<http://www.iam.unibe.ch/~scg/>

Abstract

Developing applications that end users can customize is a challenge, since end users are domain experts but still have concrete requirements. In this master thesis we present how we used a meta-driven approach to support the end user customization of Web applications. We present Magritte, a recursive meta-data meta-model integrated into the Smalltalk reflective meta-model. The adaptive model of Magritte enables to not only describe existing classes but also let end users build their own meta-models on the fly. Further on we describe how meta-interpreters automatically build views, reports, validating editors and persistency mechanisms.

As a complete example of how we applied a meta-model to a Web application we present Pier, the second version of a fully object-oriented implementation of a content management system and Wiki engine. Pier is implemented with objects from the top to the bottom and is designed to be customizable to accommodate new needs. The integration of a powerful meta-description layer makes it a breeze to extend the running system with new functionality without having to patch the core engine.

We describe the lessons learned from using the Magritte meta-model to build applications. Both projects described in this thesis are open source and can be downloaded from the Web site of the author.

Acknowledgements

First I wish to thank my supervisor Prof. Dr. Stéphane Ducasse for his guidance and that he motivated me to learn Smalltalk and join the Software Composition Group. It was a great experience to travel with him to different places around Europe and to give presentations about my work.

I would like to thank Prof. Dr. Stéphane Ducasse and Prof. Dr. Roel Wuyts for the discussions on the design and implementation of SmallWiki, Pier and Magritte. Thanks for writing with me the two papers, [Duca05] and [Reng07], which were an important point of reference for this master thesis.

Also I would like to thank Prof. Dr. Oscar Nierstrasz, head of the Software Composition Group, for giving me the opportunity to work in his group, for the careful reading of this master thesis and the constructive comments that helped me to improve it.

I would like to thank my parents for all their support and encouragements during my studies. I also would like to express my thanks to all my friends.

Lukas Renggli
June 2006

Contents

Abstract	iii
Acknowledgements	v
Contents	vii
1 Introduction	1
1.1 Approach	1
1.2 Outline	2
2 Magritte	3
2.1 Context and Constraints	3
2.2 Describing Domain Objects	4
2.3 Interpreting Descriptions	7
2.3.1 Building Textual Views	7
2.3.2 Object Relational Mapping	8
2.3.3 Building Validating Editors	8
2.3.4 Customizing the Meta-Interpretation	10
2.4 Meta Magritte	11
2.4.1 Adaptive Model	11
2.5 Aare	13
2.6 Implementation	14
2.6.1 Descriptions	15
2.6.2 Accessors	15
2.6.3 Conditions	17
2.6.4 Mementos	18
3 Pier	19
3.1 Introduction	20
3.2 History	21
3.3 Pier in Action	23
3.4 Architecture	24
3.4.1 Separation of Concerns	24

3.4.2	Pages and Files	25
3.4.3	Visitors	28
3.4.4	Context and Commands	30
3.4.5	Environment	32
3.5	Extending Pier	33
3.5.1	Fixing broken links	33
3.5.2	Converting documents	34
3.5.3	Security	35
3.6	Pier at the Meta-Level	36
3.6.1	Searching	37
3.6.2	Persistency and Versioning	38
3.6.3	Adaptive Forms	39
3.7	Lessons Learned	43
3.8	Summary	45
4	Conclusion	47
4.1	Related Work	48
4.2	Lessons Learned	51
4.3	Further Work	52
	Documentation	53
	Magritte-Model-Core	53
	MACompatibility	53
	MADistribution	53
	MAObject	53
	Magritte-Model-Models	55
	MAAdaptiveModel	55
	MAFileModel	56
	MATableModel	57
	Magritte-Model-Description	57
	MABooleanDescription	57
	MAClassDescription	58
	MAColorDescription	58
	MAContainer	58
	MADateDescription	58
	MADescription	58
	MADurationDescription	63
	MAElementDescription	63
	MAFileDescription	64
	MAMagnitudeDescription	64
	MAMemoDescription	64
	MAMultipleOptionDescription	65
	MANumberDescription	65
	MAOptionDescription	65

MAPasswordDescription	65
MAPriorityContainer	66
MAREferenceDescription	66
MARElationDescription	66
MASingleOptionDescription	67
MAStringDescription	67
MASymbolDescription	67
MATableDescription	67
MATimeDescription	68
MATimeStampDescription	68
MATokenDescription	68
MAToManyRelationDescription	68
MAToOneRelationDescription	68
Magritte-Model-Accessor	68
MAAccessor	69
MAAutoSelectorAccessor	69
MABlockAccessor	69
MAChainAccessor	70
MAContainerAccessor	70
MADictionaryAccessor	70
MANullAccessor	70
MASelectorAccessor	70
MAVariableAccessor	71
Magritte-Model-Condition	71
MAAllCondition	71
MAAnyCondition	71
MACondition	71
MAFalseCondition	72
MANoneCondition	72
MASelectorCondition	72
MATrueCondition	73
Magritte-Model-Memento	73
MACachedMemento	73
MACheckedMemento	73
MAMemento	74
MAStraitMemento	74
Magritte-Model-Exception	74
MAConditionError	74
MAConflictError	74
MAError	75
MAKindError	75
MAMultipleErrors	75
MARangeError	75
MAREadError	75

MARequiredError	76
MAValidationError	76
MAWriteError	76
Magritte-Model-Visitor	76
MAVisitor	76
Magritte-Model-Utility	77
MADynamicObject	77
MANamedBuilder	77
MAPragmaBuilder	78
MAProxyObject	78
Pier-Model-Core	78
PObject	79
Pier-Model-Kernel	80
PRContext	80
PRCurrentContext	81
PRKernel	81
Pier-Model-Structure	82
PRChildren	82
PRDecorated	83
PRDecoration	84
PRFile	85
PRHider	85
PRPage	85
PRStructure	86
Pier-Model-Document	89
PRAnchor	89
PRDocument	89
PRDocumentGroup	89
PRDocumentItem	90
PRDocumentParser	90
PRDocumentWriter	91
PRExternalLink	91
PRHeader	91
PRHorizontalRule	92
PRInternalLink	92
PRIsbnLink	92
PRLink	93
PRList	94
PRListItem	94
PRMailLink	94
PROrderedList	94
PRParagraph	94
PRPreformatted	94
PRRfcLink	95

PRTable	95
PRTableCell	95
PRTableRow	95
PRText	95
PRUnorderedList	96
Pier-Model-Command	96
PRCommand	96
Pier-Model-Visitor	98
PRFullTextSearch	99
PRIncomingReferences	99
PROutgoingReferences	99
PRPathLookup	100
PRPathReference	100
PRVisitor	101
Index	102
Bibliography	107

Chapter 1

Introduction

“I would rather write programs to help me write programs than write programs.”

— Dick Sites

Many applications consist of a large number of input dialogs and reports that need to be built, displayed and validated manually. Often these dialogs remain static after the development phase and cannot be changed unless a new development effort occurs. End users often need to rapidly adapt their applications to new business needs [Yode02]. In many cases they would know how to make the required adaptations if the application would let them do so [Atki87].

For certain kinds of application domains such as small-businesses, changing business plans, modifying workflows, etc. usually boils down to minor modifications to domain objects and behavior, for example new fields have to be added, configured differently, rearranged or removed. Unfortunately most of today’s applications don’t provide this flexibility to their end users. The situation is even more striking for 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.

1.1 Approach

Our solution to this problem is to describe domain objects with Magritte, a self-described meta-model, and to provide a framework that interprets this

meta-model in different ways, for example to display, manipulate, validate and store domain objects. Magritte is simple enough that end users are actually able to modify or extend the existing meta-model to make the application fit their exact needs. Moreover Magritte is self-described, enabling the automatic generation of meta-editors. Finally Magritte is integrated in the Smalltalk language which serves as an executable meta-language [Clar04, Mull05a].

Magritte is an adaptive object model implementation that does not concentrate on a specific domain as suggested in [Rieh05, Yode02], but is more generic and can be potentially used in any software project. The Magritte meta-model is powerful enough that application developers can specify how their domain objects are structured, how they should be stored, and how they can be modified so that views, editors and reports can be built automatically.

1.2 Outline

- [Chapter 2](#) introduces Magritte, explains the basic usage of the framework and discusses important points about its implementation. Moreover it presents use-cases and examples where the framework has successfully been applied to.
- [Chapter 3](#) presents Pier, a meta-described content management and Wiki system. This chapter will illustrate the use of Magritte in a wider context and the lessons learned while developing a large meta-described Web application in two iterations.
- [Chapter 4](#) we will conclude our experience while implementing and using Magritte in large projects, such as Pier. It will compare our approach with related work and identify future work.
- The [Appendix](#) will provide a complete class documentation of Magritte and Pier, automatically generated from the source code of the respective projects.

Chapter 2

Magritte

“René Magritte: A consummate technician, his work frequently displays a juxtaposition of ordinary objects, or an unusual context, giving new meanings to familiar things. The representational use of objects as other than what they seem is typified in his painting, La trahison des images, which shows a pipe that looks as though it is a model for a tobacco store advertisement. Magritte painted below the pipe, ceci n’est pas une pipe, which seems a contradiction, but is actually true: the painting is not a pipe, it is an image of a pipe.”

— Wikipedia, <http://en.wikipedia.org/wiki/Magritte>

This chapter is structured as follows: [Section 2.1](#) presents the context and the constraints that influenced Magritte. [Section 2.2](#) introduces the Magritte framework. [Section 2.3](#) presents various examples of how Magritte descriptions can be interpreted. [Section 2.4](#) explains how Magritte is self-described and how this enables end users to customize their applications. [Section 2.5](#) gives a concrete example how we applied Magritte in a Web based workflow engine.

2.1 Context and Constraints

As a result of our experience with developing complex Web applications we recognized the need to introduce a meta-layer to provide us with more flexibility. However, the meta-layer has to cope with the constraints and the context of our development.

For our Web development we are using Seaside [[Seaside](#)], a framework combining an object-oriented approach with a flow-based one. Seaside gives

us key advantages over traditional page-centric approaches [Frat99], as it represents pages as a set of collaborating components or objects which do not need taking into consideration HTTP constraints. With Seaside, all the development tools (versioning, navigation, testing, debugging) behave as if we were developing a desktop application, for example Seaside Web applications can be as easily debugged as any Smalltalk application:

- Hot-Debugging: The debugger can inspect, modify and send messages to objects on the fly.
- Hot-Recompilation: Methods can be changed and added while the application is running.
- Dynamic Code Reloading: The application can be updated while the server is running.

Since applications developed using our meta-model should be extended or changed (maintainability), the introduction of meta-descriptions should not disrupt the normal way and the tools used to program. In particular, Seaside and plain object-oriented programming should be possible. Generative techniques should be avoided, as these make it difficult to maintain and change the code later on. Moreover generative techniques prevents one from dynamically change the meta-model at runtime. The development tools (refactorings, versioning, navigation, testing, ...) should continue to work as if there would be no meta-descriptions. For all these points, the approach should be integrated as close as possible into the object-oriented paradigm, the tools and the programming environment. In our case we use Squeak, an open-source Smalltalk [Inga97].

The solution we describe next is based on meta-descriptions which are tightly integrated in the Smalltalk reflective architecture [Riva96].

2.2 Describing Domain Objects

Describing domain entities is not a new idea. Object-oriented meta-languages such as MOF [Grou97], EMOF [Grou04] or ECore [Budi03] are often used to describe domain specific language meta-models. However, such object-oriented meta-languages only support the structural description. They do not have support for the definition of behavior and as such cannot be used to specify the operational semantics of meta-models [Mull05a]. Magritte is a meta-description framework, describing domain classes and their respective fields and relationships. Magritte is integrated in the Smalltalk meta-model. Smalltalk is used to define Magritte meta-entities and their behavior. A field description contains the type information, the way the field is accessed, some optional information such as a field

comment and label, relationships and validation conditions; furthermore it defines Boolean properties, such as if the field is required, read-only, visible, persistent, etc.

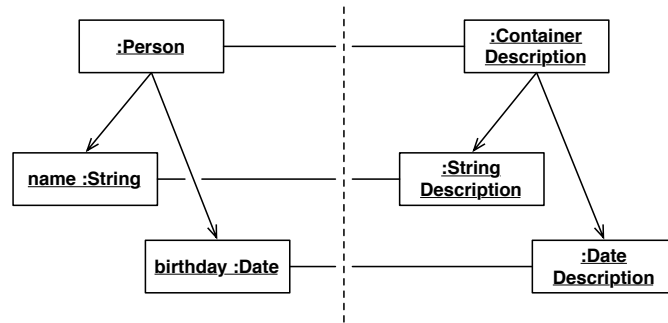


Figure 2.1: Person model instance (left) with associated descriptions (right).

Example. A described instance of a person domain-model, with the fields **name** and **birthday**, could look like Figure 2.1. To describe the three entities in this model we need three corresponding description instances, that can be built statically at design-time, dynamically at run-time, or even a combination of the two. Either-way, the code to create those description instances looks exactly the same: to describe the **name**, we give it an access-selector, a label, and we tag it as a required value¹.

```
(StringDescription selector: #name label: 'Name')
  beRequired;
  yourself
```

Note that statically typed languages could provide some clues about what is stored within instance variables using their introspection facilities. In Smalltalk there is no static-typing, therefore the field **birthday** could point to any kind of object, *e.g.*, a date, a time-stamp, a number or even a string object. Nevertheless typing does not solve this problem, since types do not tell us how the value should be displayed (June 11, 1980, 11 June 1980, 06/11/1980), edited (text-input fields, drop-down boxed, date-picker), saved or validated. The following description definition looks similar to the one above but adds a different validation condition: the birthday is not required but if a date is given, it has to be between 1900 and today:

```
(DateDescription selector: #birthday label: 'Birthday')
  between: (Date year: 1900) and: Date today;
  yourself
```

¹Note that all the code expressions given here can be evaluated and inspected by copying them to a Smalltalk workspace.

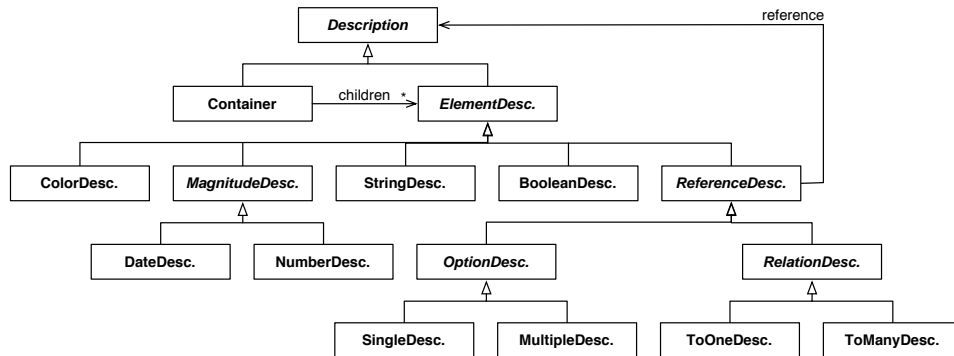


Figure 2.2: The description hierarchy is a composite of meta-entities.

Descriptions, as we have seen in the above examples, are naturally organized in a description hierarchy. A class diagram of the most important descriptions is shown in Figure 2.2. Different kinds of descriptions exist: simple type-description that directly map to Smalltalk classes, and some more advanced descriptions that are used to represent a collection of descriptions, or to model relationships between different entities.

Type Descriptions. Most descriptions belong to this group, such as the `ColorDescription`, the `DateDescription`, the `NumberDescription`, the `StringDescription`, the `BooleanDescription`, etc. All of them describe a specific Smalltalk class; in the examples given, this would be `Color`, `Date`, `Number` and all its subclasses, `String`, and `Boolean` and its two subclasses `True` and `False`. All descriptions know how to perform basic tasks on those types, such as to display, to parse, to serialize, to query, to edit, and to validate them.

Container Descriptions. If a model object is described, it is often necessary to keep a set of other descriptions within a collection, for example the description of a person consists of a description of the title, the family-name, the birthday, etc. The `ContainerDescription`, and its subclasses, provides a collection container for other descriptions. In fact the container implements the whole collection interface, so that users can easily iterate (`#do:`), filter (`#select:`, `#reject:`), transform (`#collect:`) and query (`#detect:`, `#anySatisfy:`, `#allSatisfy:`) the containing descriptions.

Option Descriptions. The `SingleOptionDescription` describes an entity, for which it is possible to choose up to one item from a list of objects.

The `MultipleOptionDescription` describes a collection, for which it is possible to choose any number of items from a predefined list of objects. The selected items are described by the referencing description.

Relationship Descriptions. Probably the most advanced descriptions are the ones that describe relationship between objects. The `ToOneRelationshipDescription` models a one-to-one relationship; the `ToManyRelationshipDescription` models a one-to-many relationship using a Smalltalk collection. In fact, those two descriptions can also be seen as basic type descriptions, since the `ToOneRelationshipDescription` describes a generic object reference and the `ToManyRelationshipDescription` describes a collection of object references. The referenced objects are described by the referencing description, which is – if not manually defined by the developer – automatically built from the intersection of the element descriptions.

2.3 Interpreting Descriptions

Having described domain objects opens up a number of different possibilities by writing meta-interpreters that walk over the descriptions and perform different tasks on the model. The most immediate is that descriptions are used to automatically build views, editors and reports.

2.3.1 Building Textual Views

The simplest interpreter that can be written is one that iterates over all descriptions of a domain model and prints the labels 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 `aModel`:

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

First we ask the model for its container-description, then we iterate over its individual description elements. 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 we can append it to the output.

Since every description knows how to print its values, 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 some values are printed, for example if we want to print dates with the months name written out. When we are adding, removing or changing descriptions in the domain-model, the above code will still print the correct output without us having to change a single line of the interpretation code.

2.3.2 Object Relational Mapping

In a very similar way, we are able to automatically create SQL statements to store, load and query objects from a relational database. Since the descriptions of Magritte can be directly mapped to an entity-relationship model, it is straightforward to define such interpreters:

- Container-descriptions map to tables, with a primary key to uniquely identify the objects in the database and the containing descriptions as attributes.
- Type-descriptions map to attributes of the appropriate SQL data-types: `BooleanDescriptions` map to `BOOLEAN` attributes, `StringDescriptions` map to `VARCHAR` attributes, etc.
- Relationship-descriptions map to a foreign key of a different table. Depending on the cardinality of the relationship an intermediate linking table is automatically introduced.

The strength of this approach is that we are not forced to embed SQL into our application code: changes to persistent objects are automatically handled by Magritte and propagated to the database. Simple changes of the descriptive model, such as adding, removing or changing descriptions, trigger a transparent migration of the database. Moreover the database back-end can be changed anytime, a customized SQL code generator will take care of the differences in the dialects.

2.3.3 Building Validating Editors

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 interface frameworks, as seen in [Figure 2.3](#).

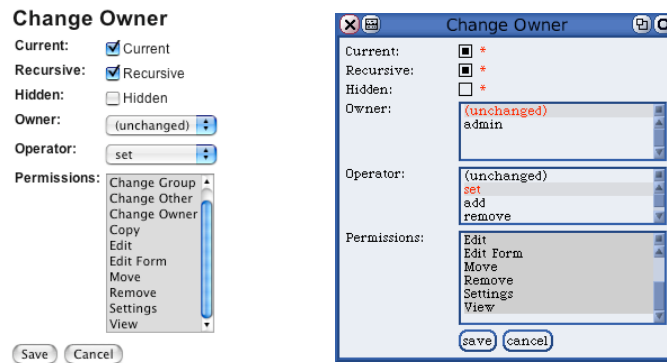


Figure 2.3: Interpreting descriptions for different GUI frameworks: the Web (left) and Morphic Squeak [Sque10] (right)

Sending the message `#asComponentOn:` to a description with a domain model as argument returns a ready-to-use Seaside [Seaside] component that can be plugged into a Web application. For convenience one might also send the message `#asComponent` to the domain model and let Magritte figure out the required descriptions itself. As in Section 2.3.1, Magritte will iterate over the descriptions and compose an editor from the collection of descriptions.

During an edit operation, Magritte works on a copy of the model, so that the original model remains in a valid state all the time and the edit operation can simply be cancelled by closing the editor window or hitting the cancel button. The original object is never touched until the edited model satisfies all its validation conditions. Moreover, before committing the changes to the actual object, Magritte checks that there are no edit conflicts caused by other people editing the same objects at the same time, and, if necessary, offers to merge those changes. The *unit of work* during an edit operation is the described object. Changes to other objects are not automatically tracked by Magritte.

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. All kinds of editing concerns are handled at only one place and not duplicated across all editors in the system.

2.3.4 Customizing the Meta-Interpretation

Metadata driven architectures are ideal for supporting meta-tools and as such let the programmer automate cumbersome tasks such as building input forms, editors, and serializer. However, they often hamper the fine-grained customization of the resulting elements such as widgets. For Magritte we paid attention not to enclose the developer within a specific interpretation of the description, but to give him or her the possibility to customize any part of the editor building process:

Custom Rendering. The default builder puts the edit widgets from top to bottom with the labels on the left side, as seen in [Figure 2.3](#). Sometimes other layouts are more convenient, for example the widgets should be laid out from left to right with the labels on top, or they should appear within other user interface elements that are maybe not under the control of Magritte. In a Web context different style-sheets can sometimes help to achieve the desired effect, however there are examples where this doesn't help or is simply too cumbersome. Magritte allows one to define one's own builder by subclassing a Visitor and overriding some of the methods used to place the user interface elements.



Figure 2.4: Different custom widgets for a single-selection description (left) and a multi-selection description (right).

Custom Widgets. Magritte tries to guess which widgets suits the description best, such as a text-input field for a string description or a checkbox for a Boolean description. However there are cases where different widgets make sense, such as with a single-selection description, that could be displayed as a drop-down box or as an radio-group, see [Figure 2.4](#). Magritte provides developers with the choice among a vast collection of possible widgets, and even gives the possibility to provide custom widgets that behave exactly the way it is required.

2.4 Meta Magritte

Magritte is integrated into Smalltalk, where everything is an object. This means that any Smalltalk object can be described using Magritte, either by a primitive type description such as `String`, `Boolean` or `Number`, or for composed objects by a composite description. The reflective facilities of Smalltalk are cleanly enhanced with those provided by Magritte.

It is natural that we apply descriptions recursively, therefore descriptions are also described. As seen in [Figure 2.5](#), there is an optional association to a set of descriptions from the root of the class hierarchy in `Object`. The default implementation in `Object` returns an empty description container, however subclasses usually add their own specific descriptions to accommodate their needs, and `Description` is such a class.

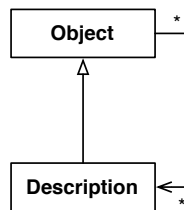


Figure 2.5: Descriptions as described objects.

2.4.1 Adaptive Model: Enabling End User editable Meta-Descriptions

Having a meta-described framework makes it possible to let end users create and edit their meta-models on the fly. To accommodate this need we created a generic object model called `AdaptiveModel`, mapping descriptions to actual values, as seen in [Figure 2.6](#). The `AdaptiveModel` has two instance variables, the first being used to refer to the description of the model instance and the other one to keep a list of the actual values of the model.

The user is able to edit the adaptive model at two different levels, at the meta-model and at the model level:

Meta-Level 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

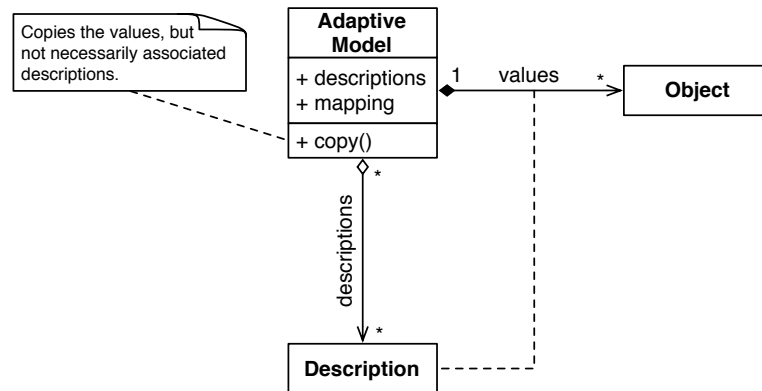


Figure 2.6: An adaptive model, mapping a set of descriptions to actual model values.

using themselves, an editor allows one to modify the descriptions of the model itself can be built automatically, see Figure 2.7.

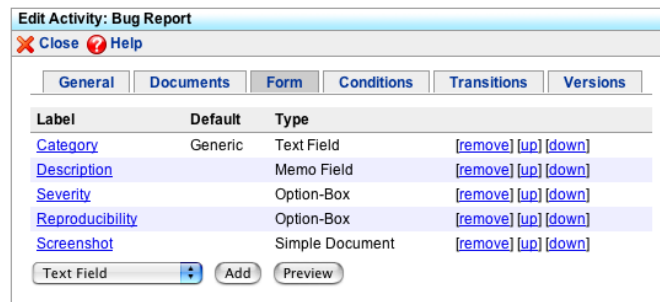


Figure 2.7: A Magritte description editor for the description of an adaptive model.

Model-Level Editing. Since the adaptive model is described, it can be edited as shown in Section 2.3.3. 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, as seen in Figure 2.6. This gives much better flexibility when descriptions are changed. The resulting editor in Figure 2.8 looks the same as if the descriptions were defined statically.

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 meta-model affects all its associated instances:

Figure 2.8: A Magritte editor for the adaptive model.

- Adding a new description creates a new attribute with the default value specified in the description.
- Editing existing descriptions propagates to all existing attributes. Note that most edit operations on descriptions do not affect the validity of the actual value, such as to change the label or the default value. However there might be changes that change validation conditions and that might turn existing models invalid. Our adaptive model doesn't fix those invalid instances automatically, either the programmer has to treat the problem manually or at the next edit operation the user will be notified by the validation code of Magritte and has to fix the problem.
- Removing an existing description removes the associated values.

2.5 Aare: A Workflow Definition- and Runtime-Engine with adaptive Forms

Figure 2.7 shows a screenshot of a workflow definition engine that we implemented using Seaside [Seaside, Duca04] and Magritte. It allows end users to specify their own forms for their workflow activities: the drop-down box offers a list of possible description prototypes that can be added to the form definition, such as text, memo, number, date, time and money fields, check, and option boxes, and some special fields such as uploaded documents or tables.

Since not all the descriptions are meaningful to the end users or too implementation specific, we do not allow end users to add all the available

descriptions in Magritte, but only a small selection of commonly used descriptions. Moreover we do not offer the possibility of editing every property of a description, again to reduce the complexity of the application: when adding or editing descriptions only a subset of the available properties are displayed, such as the label, a comment, the default value and some other description specific editors.

Figure 2.9 shows the 'Edit Activity: Bug Report' dialog box, specifically the 'Conditions' tab. The dialog has a title bar with 'Edit Activity: Bug Report' and buttons for 'Close' and 'Help'. Below the title bar are tabs for 'General', 'Documents', 'Form', 'Conditions' (selected), 'Transitions', and 'Versions'. The main area contains the text 'Satisfy if all (change to any) of the following conditions are met:'. Below this is a table with three columns: 'Field', 'Comparison', and 'Value'. The table contains two rows: one for 'Category' with comparison 'is not' and value 'blank', and another for 'Screenshot' with comparison 'is not' and value 'blank'. Each row has a '[remove]' link. Below the table, there is a 'Rule on field' dropdown set to 'Category (Text Field)' and an 'Add' button.

Field	Comparison	Value
Category	is not	blank [remove]
Screenshot	is not	blank [remove]

Rule on field: Category (Text Field) [Add]

Figure 2.9: Defining the validation-conditions.

Validation conditions for form fields are defined in a different part of the editor, as seen in Figure 2.9. In the given example the field *category* has to be filled and a screenshot must be uploaded to validate the form. Depending on the underlying description the condition editor displays a set of usable conditions that can be added to or removed from the list, for example for a text field these are *is*, *is not*, *begins with*, *ends with*, *matches*; for a number field these are *is*, *is not*, *greater than*, *smaller than*; etc. Prototype conditions (*i.e.*, *is not*, *greater than*) that are themselves described by descriptions, are defined by every description class (*i.e.*, `StringDescription`, `DateDescription`, ...), so that an editor for all the conditions can be built and displayed automatically.

In Figure 2.8 the defined form is shown as it is presented to the end user at runtime of the workflow engine. To allow people to save activities that are in progress and that have failed validation conditions, we decided to strictly separate editing and validation: the form can be saved any time and is automatically validated as seen below, however the activity can only be completed if the conditions all validate.

2.6 Implementation

Magritte consists of a collection of packages. In the following sections we describe the responsibilities of the most important packages and show how they relate to each other.

2.6.1 Descriptions

The description hierarchy, see [Figure 2.2](#), plays a central role in Magritte. It provides the different description types, as explained in [Section 2.2](#):

- “Type descriptions” describe specific Smalltalk classes. They are primitive entities in Magritte and do not delegate to other descriptions. Their implementation is straightforward and often directly maps to methods already present in the Smalltalk class library, such as to parse numbers or to serialize strings.
- “Container Descriptions” describe collections of descriptions. Often it is necessary to store descriptions in a specific order or to group them, for example a model is usually described by a single container description that references a collection of other descriptions for each of its instance variables. Containers understand the collection protocol as known from the Smalltalk class library.
- “Option Descriptions” describe attributes for which one or more items out of a list of objects can be selected. The objects to choose from are described by the reference description. Option descriptions can potentially be extended by end users with custom options at runtime.
- “Relationship Descriptions” describe relationships between objects. A relationship is always defined from the described object to the referenced object. To describe a two-way relationship the developer has to define a relationship description at both ends of the association.

2.6.2 Accessors

In Smalltalk data can be accessed and stored in different ways. Most common data is stored within instance variables and read and written using accessor methods, but sometimes developers choose other strategies, for example to group data within a referenced object, to keep their data stored within a dictionary, or to calculate it dynamically from block closures.

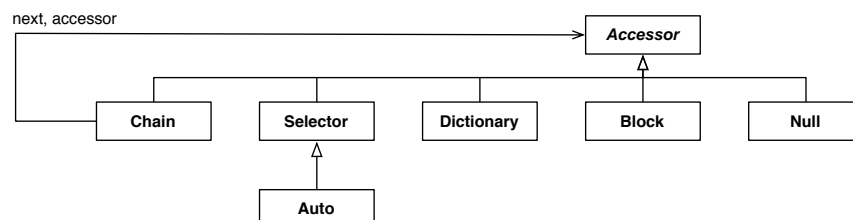


Figure 2.10: The accessor is a strategy on how to access model data

To give Magritte the full flexibility and a transparent way to access different data sources, we provide a Strategy pattern [Alpe98, Page 339] as seen in Figure 2.10. By far the most commonly used accessor type is the **SelectorAccessor**. It can be instantiated with two selectors: a zero argument selector to read, and a one argument selector to write. For convenience it is possible to specify a read selector only, from which the write selector is inferred automatically.

A special form of the **SelectorAccessor** is the **AutoSelectorAccessor**: it automatically creates read accessors, write accessors, and instance variables if necessary. This is very useful for fast prototyping, if the model classes haven't been fully specified yet; at a later stage of development this accessor can be easily replaced with a **SelectorAccessor**.

The **DictionaryAccessor** is used to add and retrieve data from a dictionary with a given key. This access strategy is also mainly used for prototyping as it allows one to treat dictionaries like objects with object-based instance variables.

The **ChainAccessor** is used to build a sequence of two or more access strategies. To read and write a value the **accessor** is evaluated on the given model and the result is passed into the **next** accessor.

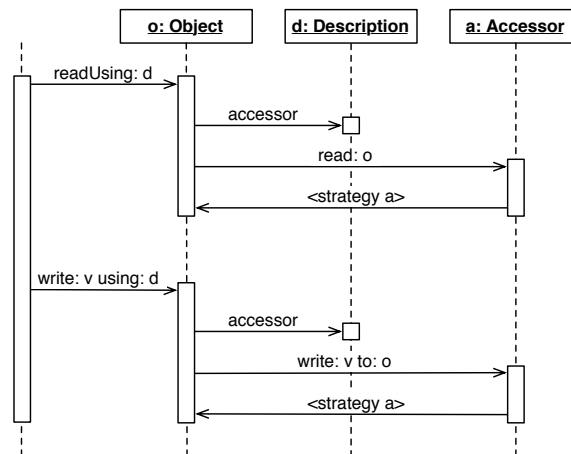


Figure 2.11: Reading from and writing to a described model

As visualized in Figure 2.11 all parts of Magritte access data by dispatching it through the model using the description and its associated accessor. This allows one to reify or intercept access from the model, before it is actually performed by the accessor instance. Being able to reify model access opens a lot of new possibilities, such as to provide a security layer, or to dispatch data access to other sources, *e.g.*, external resources (internet, database, file system).

2.6.3 Conditions

Applications often require a lot of slightly different validation strategies. As an example one would like to store an e-mail address as a string. Obviously the `StringDescription` would be the perfect choice, but you would need to add conditions, so that the input has to match a specific pattern and that it is enforced to belong to a Swiss domain. To avoid an unlimited growth of the description hierarchy by creating subclasses for every new validation strategy, it is possible to provide block closures that serve as additional validation strategies:

```
(StringDescription selector: #email label: 'E-Mail Address')
  addCondition: [ :value |
    (value matches: '**@#*.#*')
    and: [ value endsWith: '.ch' ] ]
  label: 'Invalid E-Mail';
  yourself
```

The problem with this approach is that in Smalltalk block closures are hardly serializeable, since they might reference globals and other variables within their execution context. To avoid this problem `#asCondition` can be sent to the block closure to turn it into a composite of condition objects that can be easily stored to and loaded from external sources. This is currently possible for simple expressions only and if there are no external references. It works, however, in most of the common cases where the value is the receiver of some messages to check its validity.

```
(StringDescription selector: #email label: 'E-Mail Address')
  addCondition: [ :value |
    (value matches: '**@#*.#*')
    & (value endsWith: '.ch') ] asCondition
  label: 'Invalid E-Mail';
  yourself
```

Of course it is also possible to manually build condition objects, either through an end user interface or by writing code that instantiates and composes the condition objects:

```
(StringDescription selector: #email label: 'E-Mail Address')
  addCondition: (AllCondition
    with: (SelectorCondition
      selector: #matches:
      arguments: #('**@#*.#*'))
    with: (SelectorCondition
      selector: #endsWith:
      arguments: #('.ch')))
  label: 'Invalid E-Mail';
  yourself
```

In fact, all the tree source examples above answer a description that shows the same validation behaviour, however the second and the third one only are serializeable. When giving preference to readability and serializeability to choose the second approach would probably be preferable.

2.6.4 Mementos

The Memento design pattern [Alpe98, page 297] records the state of an object so that it is possible to delay changes, to detect changes to an object or to restore the object to its original state later. Most users of Magritte don't need to know about the Memento pattern which is only used internally to cache model state. However the mementos play a central role if a developer wants to fully understand the internal workings of the framework, especially together with the automatic building of user interfaces.

Editing a model object might temporarily invalidate it: this means that model invariants could be invalid, and not all the built-in or manually added conditions are satisfied all the time. Especially if multiple users concurrently work on the same model it is important to always ensure the consistency of the model. Moreover people might decide to cancel edit operations, which should turn back the model state exactly to the point it was before the edit operation was started.

The memento hierarchy in Magritte provides classes that behave – from the perspective of Magritte – like the corresponding original model and that delay modifications until they are proven to be valid. Modifications can then be committed in a controlled transaction, so that concurrent changes can be detected and conflicts can be sorted out and eventually merged. Finally the cached data can be stored into the model.

Since it might be required to tweak the behaviour of the default memento in some cases, for any Smalltalk object the method `#mementoClass` can be overridden to return a different memento class. This can be especially useful for databases that require one to update, commit or tag modified objects, which can be efficiently done within a memento subclass after committing the changes to the persistent object.

Chapter 3

Pier

“A pier is a raised walkway over water, supported by widely-spread pillars. Today the most common form of a pier is the industrial pier which can be found at ports throughout the world. A pier may be open air or closed. Sometimes a pier has two decks.”

— Wikipedia, <http://en.wikipedia.org/wiki/Pier>

This chapter presents Pier, a second version of a fully object-oriented implementation of a meta-described content management¹ and Wiki² system. Pier still inherits a lot of Wiki functionality from its first version called SmallWiki. Over the years Pier has grown into a full-fledged application and content management framework. It is written with objects from the top to the bottom and it can be customized easily to accommodate new needs. Pier is based on Magritte to enable the building of all the user interface elements declaratively, and to enable sophisticated search queries and persistency.

This chapter is structured as follows: [Section 3.1](#) gives an introduction of the problems of current Wiki implementations and why we decided to develop a new one. [Section 3.2](#) presents the history and evolution of Pier. [Section 3.3](#) reveals several real world examples that make use Pier in productive environments. [Section 3.4](#) explains the core architecture of Pier and the most important design decisions. [Section 3.5](#) gives examples of how to extend the system with new functionality. [Section 3.6](#) presents the benefit of having a

¹A content management system is software to organize and edit documents. Most of the time editing is done through a Web interface by dedicated editors.

²A Wiki is a Web site that allows anybody to add and edit content collaboratively, mostly without requiring registration. The term Wiki also refers to the software that helps to create such a Web site.

descriptive meta-model for its implementation. [Section 3.7](#) finally presents the lessons learned while implementing a content management system in two major iterations.

3.1 Introduction

While Wiki systems offer a significant degree of freedom to their users to edit and share content [[Leuf01](#)], the underlying implementations are often less flexible and powerful than the collaborative model they promote. Wiki and content management systems are mostly implemented using string-based approaches (regular expressions) to parse, generate and transform their pages. While such approaches work well for straightforward systems, they hamper the customization and adaptability of systems to the variety of end users that require more sophisticated needs, for example different output formats, user interfaces, security policies, etc.

One might think that advanced Wikis that provide functionality of content management systems are not really necessary, and hence that simple implementations that only allow users to change the contents of pages suffice. Experience shows that this is not the case:

Input and Output. Most Wikis provide users with a simple Wiki syntax to create rich XHTML pages, however they hamper the possibility to use other input and output formats. This is the reason why Pier stores the contents of a page within an abstract document tree that can be traversed to emit different output formats such as XHTML, L^AT_EX or plain text. Systems based on strings require to duplicate the parsing functionality for every new output format. For complex applications, such as Wikipedia [[WikiPedi](#)], there are so many slow regular expressions applied to the input that they are forced to implement sophisticated caching algorithms for different output and search formats.

User Interface. An experiment using Wikis in classrooms showed that children and teachers require different user interfaces and functionalities [[Duca00](#)]. Students should have a simpler user interface compared to the teachers, who should be able to lock all the pages of her students at once.

Management. Another example is maintenance, which typically requires sophisticated functionalities such as searching for all the pages containing more than 10 external links or finding all pages that were edited on a certain date and that have more than twenty uploaded pictures. Since such activities are typically done by end users themselves, they

should be supported by the system itself as not to break the metaphor of the medium.

Customizability. The metaphor of a content management system should not stop at the level of editing pages. Therefore we need a customizable application with an underlying implementation that supports the definition of new components and not only of changing the content of pages. Pier allows one to customize its look using meta-pages that can be edited just as any other page, and to include active components that provide tools enhancing the user experience, such as to display additional information and to provide navigational links.

3.2 From SmallWiki to Magritte and Pier

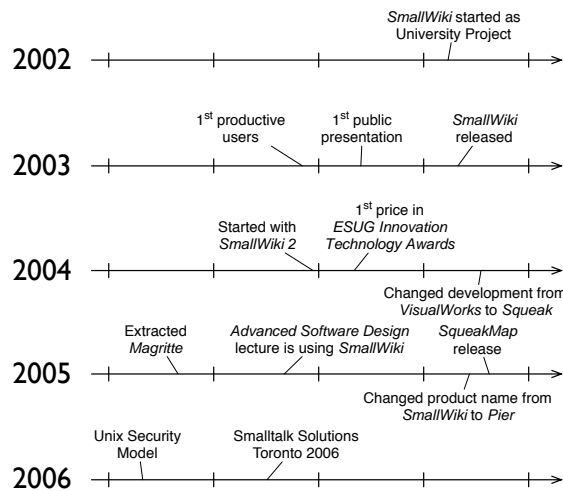


Figure 3.1: The history and evolution of Pier and Magritte

The history of Pier, as visualized in [Figure 3.1](#), started late in 2002. In the beginning the software was called SmallWiki, because its initial goal was to provide a nicely designed, fully tested and easily extensible replacement for the existing Wiki implementations in Smalltalk. A first version of SmallWiki [\[Reng03\]](#) implemented in Cincom VisualWorks was released by the end of 2003. It is still widely adopted in productive environments all over the world, as listed in [Figure 3.2](#), and is still being maintained and extended by a small group of people. SmallWiki has been ported to Squeak [\[Sque10\]](#) and #Smalltalk [\[Branb\]](#). Some other open-source frameworks such as Gaardner [\[Groo\]](#), TinyWiki and PicoWiki³ used the parser, the document

³TinyWiki and PicoWiki are both ports of a subset of SmallWiki to Seaside. The initial implementation of SmallWiki did not use Seaside as a Web server.

representation and the rendering engine for similar projects.

European Smalltalk User Group	www.esug.org
Hans Beck	www.hans-n-beck.org
Katholische Kindertagesstätte	www.kita-st-anna.de
Logo Wiki	www.logowiki.net
Lukas Renggli	www.lukas-renggli.ch
Research Center on Structural Software Improvement	restructuring.ulb.ac.be
Seaside	www.seaside.st
Squeak	www.squeak.org
Squeak Germany	www.squeak.de
Tierpark Köthen	www.tierpark-koethen.de
Tweak	tweak.impara.de
Software Composition and Decomposition (deComp)	decomp.ulb.ac.be
University of Berne	smallwiki.unibe.ch
WireSong	www.wiresong.ca

Table 3.1: Public instances of SmallWiki and Pier

By the year 2004 it became clear that the implementation of SmallWiki was lacking some important features that weren't easy to integrate into the existing model. One major problem was that the model and the view were too tightly coupled, and therefore the view could not be easily replaced with a different one. Moreover there was a lot of duplicated code that was used to generate views and editors of the model. A related problem was that some parts of SmallWiki were not easy to extend, for example it was impossible to add additional fields to a page without patching the original source code at several places. The solution to these problems was to introduce an extensible meta-layer, called Magritte.

Early in the development of Pier, it became clear that Magritte could also be useful on its own. From the very beginning, Magritte did not depend on and was not specifically targeted to SmallWiki. In spring 2005 we extracted Magritte to become its own independent framework because we needed a very similar meta-framework for a workflow system that was being developed at that time, see [Section 2.5](#). In the same year we changed the name of SmallWiki to Pier: with the integration of a meta-framework and the use of Seaside as its default view the software became much more than a wiki. Furthermore, the code base was not that small anymore.

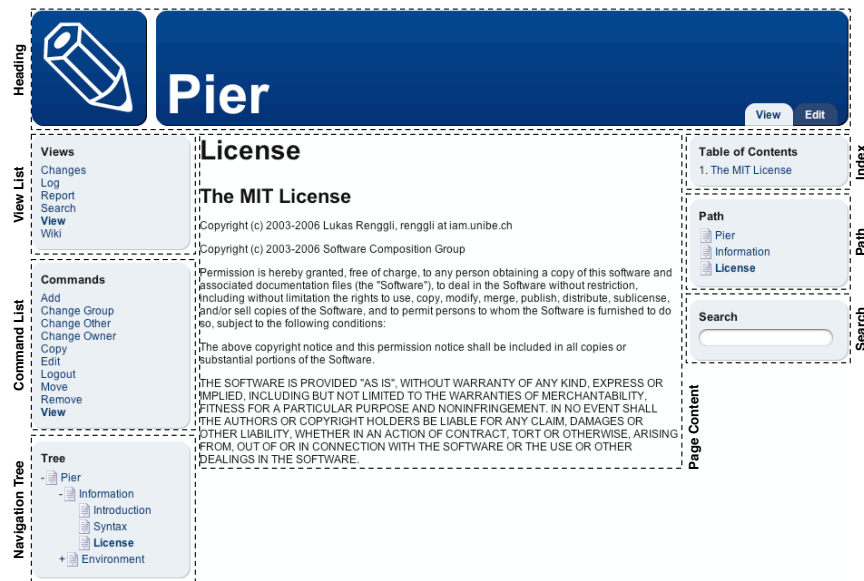


Figure 3.2: The default installation of Pier comes with a big collection of ready made components

3.3 Pier in Action

Pier structures its layout out of different components: heading, command list, navigation tree, table of contents, search interface, page content, etc. For example in Figure 3.2 the page contains the header on top and the document in the center; on the left there is a list of possible commands and a tree for easy navigation; on the right there is a table of content widget, the current navigation path and a search field.

Figure 3.3 shows a public instance of Pier where fewer components are used: the page contents, a navigation tree, a list of possible commands, and a search input box. Note that the look of Pier is based on Cascading Style Sheets (CSS), allowing the page and each component to be “skinned” differently using its CSS specification.

Logo is a simple programming language for children, used to teach the basics of computer programming. Figure 3.4 shows an instance of the “Logo Wiki” that has been built on top of Pier to enable kids to write and test Logo programs from within their Web browser. Pier takes full advantage of the wiki model that allows one to place normal paragraphs of text between the code, it makes an ideal instrument to write tutorials, and to collect and document snippets of Logo code. The “Logo Wiki” was implemented using

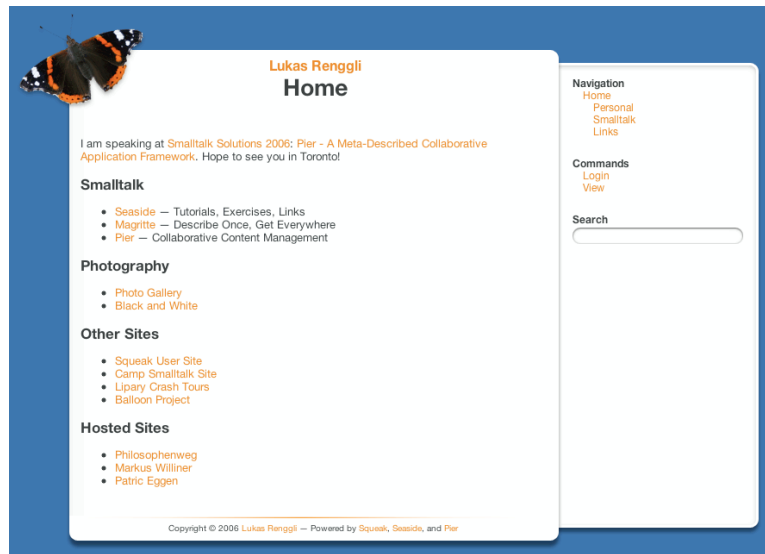


Figure 3.3: The personal Web site of the author of Pier

Pier by Luke Andrews, Avi Bryant, Andrew Catton, and Colin Putney, after an idea of Alan Kay, one of the inventors of Smalltalk.

3.4 Architecture

Pier’s design has matured over the years [Reng03]. During this process we tried to simplify it while making it more flexible. Pier has been implemented and re-implemented from scratch by the author of this master thesis. As a development environment we used Squeak, an open-source Smalltalk [Inga97]. We present here the key aspects of the implementation and the architecture of Pier.

3.4.1 Separation of Concerns

Web application development is difficult when dealing with the shortcomings of the HTTP protocol, as the right abstractions are missing [Duca04]. We therefore decided to use Seaside [Seaside] as a framework of choice for the default view in Pier. This approach greatly enhances the development of complex widgets. Since the user interface is built from Seaside components that automatically keep their state during a user session, it is easy to implement, for example, a tree-widget that is displayed on every page and remembers its expanded nodes.

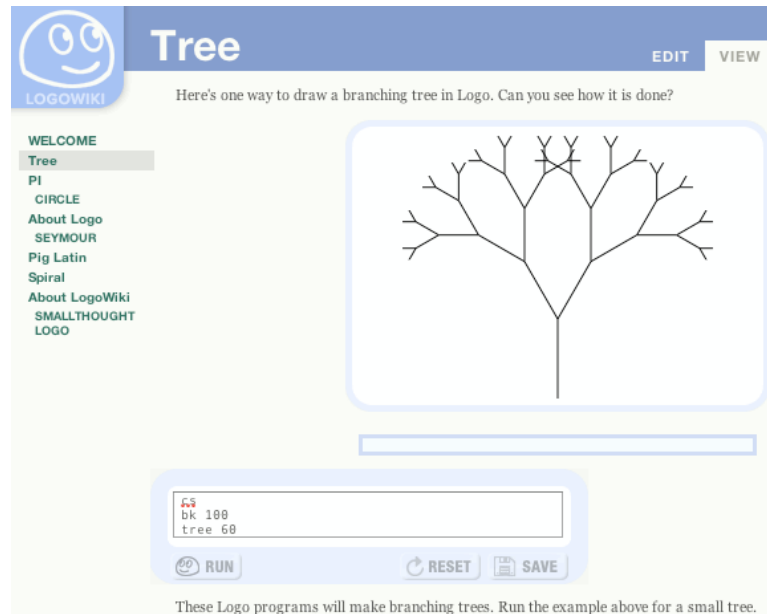


Figure 3.4: The “Logo Wiki”, a collaborative wiki to write, collect and play with Logo programs

Since Seaside offers a proper separation between the model and the view, Pier takes full advantage of it. As an example, it is possible to use a different non Web based view using the OmniBrowser [Putn] framework as shown in Figure 3.5; the same pages can be browsed and altered via the OmniBrowser interface or via a Web browser.

Another prototypical view was implemented to allow one to browse and change the model of Pier using an FTP client. It is evident that this can be useful to provide different views of the same model, depending on the location of the server and the location and abilities of the client. Having a proper meta-layer makes it possible to easily provide different views without having to duplicate the logic to create editors everywhere: for every view it is only a matter of writing a new interpreter of the meta-layer.

3.4.2 Pages and Files

As in most content management systems, structures can be nested arbitrarily within each other. Every structure consists of a unique name and a title. The Pier core implementation provides two basic structure types, pages and files, that build the main entities for any Pier site. A page is a structure containing a document, that can be edited using the wiki syntax described

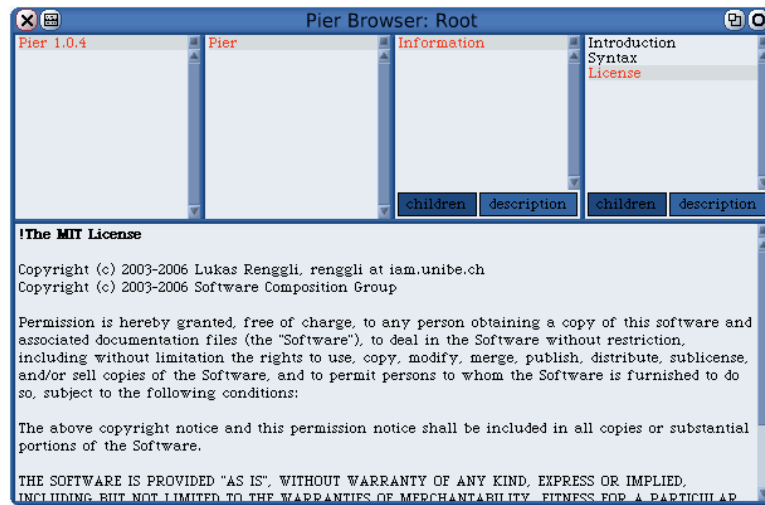


Figure 3.5: OmniBrowser view on a Pier model

below. A file is a resource that has been uploaded, for example an image, video, sound or PDF file. Other page types might be available depending on the current view (view specific structures) or if extensions to the base package have been loaded.

Pages reference a document representing their contents. The document is a Composite [Alpe98, page 137] and includes all the basic elements to represent text, such as paragraphs, ordered and unordered lists, tables, pre-formatted texts, and links, as shown in Figure 3.6.

When the user saves a text using the wiki syntax, it is parsed using SmaCC [Braná], a compiler-compiler for Smalltalk. Only the document tree is stored within the page. A Visitor [Alpe98, page 371] walking over this tree is able to transform this composite document back into an equivalent string that the user can modify again. Some nice features, such as the possibility to align table cells and add links everywhere, even within headings, greatly enhances the uniformity of the input.

Pier supports the following wiki syntax:

Paragraph. Carriage returns are preserved, simply add a newline to begin a new paragraph.

Header. A line starting with one or more `!` becomes a header line.

Horizontal Line. A line starting with `_` (underscore) becomes a horizontal line. This is often used to separate topics.

List. Using lines starting with one or more `#` and `-`, creates a list: A block

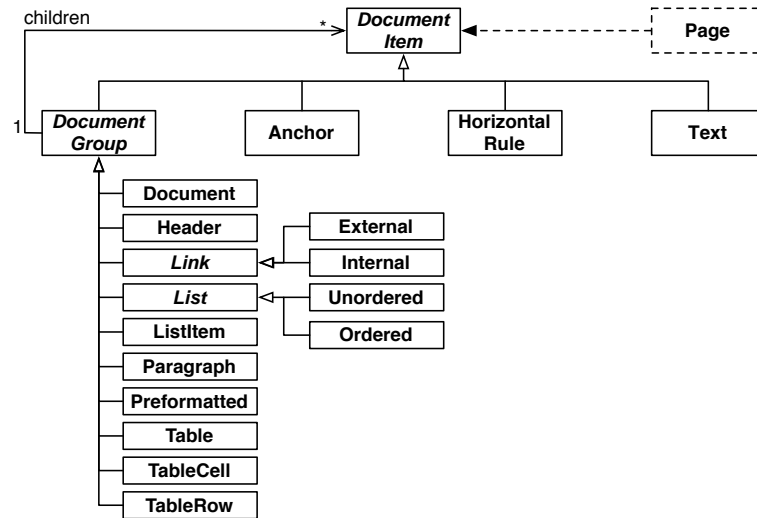


Figure 3.6: The document hierarchy

of lines, where each line starts with - is transformed into a bulleted list, where each line is an entry. A block of lines, where each line starts with # is transformed into an ordered list, where each line is an entry. Lists can be nested.

Table. To create a table, start off the lines with | and separate the elements with |s. Each new line represents a new row of the table. The contents of cells can be aligned left (default), centered or aligned right by using |{, || or |} respectively.

Pre-formatted. To create a pre-formatted section, begin each line with =. A pre-formatted section uses equally spaced text so that spacing is preserved.

Reference. To create a reference to a different structure, put it between * to create a clickable link or + to embed the reference directly into the document. All links have the form `*reference*` or `*alias>reference*`, where `reference` is in one of the following kinds:

Internal Reference. An internal reference can be written either as an absolute `*/Information/Copyright*` or relative `*../Copyright*` path. People unfamiliar with this concept will create a link without any path elements and this will reference a child of the current page, which in most cases is desired anyway. In case the path points to a non-existing structure, the user will be offered the possibility to create a new one when clicking on the link.

External Reference. If the reference is a valid URL `*http://www.-domain.com*`, a link to that external page shows up. External references cannot be embedded.

Mail Reference. If the reference is a valid e-mail address `*self@mail.me.com*`, a link to mail that person shows up. The e-mail is obfuscated to prevent robots from collecting.

ISBN Reference. If the reference is an ISBN⁴ number `*isbn:3446-202102*`, a link to the given book shows up.

RFC Reference. If the reference is a RFC⁵ number `*rfc:2616*`, a link to the given RFC page shows up.

Users are able to embed any referenced structures into the containing page, where the target is another page or file that can be embedded into the XHTML output. Note that embedding or nesting elements inside each other can lead to recursion, which when not treated correctly, would lead to infinite XHTML streams. Pier detects possible recursion problems and in case of recursion just uses link-anchors instead of embedding. Element embedding is *transparent* to the user in the sense that it is expressed using familiar syntax, for example a page with two columns is achieved by creating a table embedding two different pages each into one column of the table. This greatly enhances the possibility to build complex layouts without bloating the wiki syntax with new features or using XHTML.

Extensions have been written to enable sophisticated in-place page editing facilities in Pier: unlike most wikis, where the user is forced to edit a page as a whole entity in one big text-area, it is possible to just edit a specific paragraph that is then replaced within its context of the document with a smaller edit box. Saving that paragraph causes the text to be parsed and be merged back into the current document tree. More sophisticated editing models could also be implemented using Web 2.0 technologies, such as drag and drop and rich text editing.

3.4.3 Visitors

The implementation of Pier makes it possible to use the Visitor pattern [Alpe98, page 371] to apply operations over structures, decorations and documents. Having such a fine-grained object-oriented representation makes it very uncomplicated to implement certain features in Pier, such as to provide different input and output formats, to search for contents, or to look for broken external links, as we will demonstrate in [Section 3.5](#).

⁴International Standard Book Number, a unique identifier for books.

⁵Request for Comments, a series of numbered internet standards.

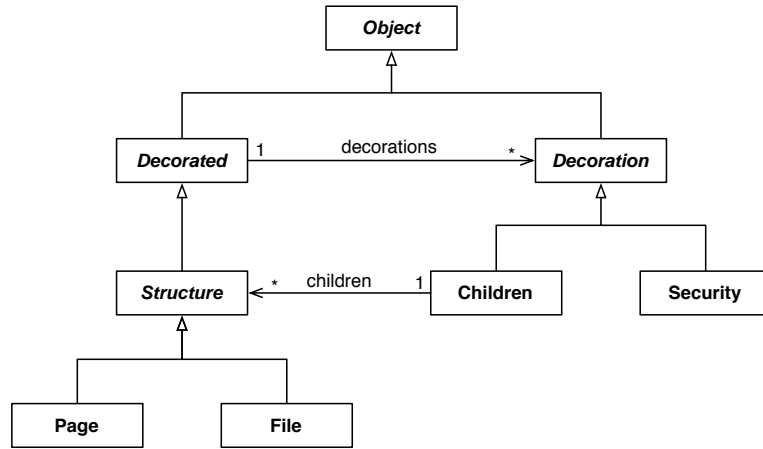


Figure 3.7: The core architecture of Pier

As seen in Figure 3.7, a `Page` or a `File` is a `DecoratedObject`, an object using a Chain of Responsibility [Alpe98, page 225] for certain aspects of its behavior, such as security or children. The decorations are tightly integrated into the Visitors so that they can easily interact with the underlying model. Decorations contain a priority that is used by the Visitor to determine the order in which the decorations are processed. The decorated object has a priority of 0. To ensure that it is processed first, the security decoration is assigned a negative priority; for details about possible security frameworks see Section 3.5.3.

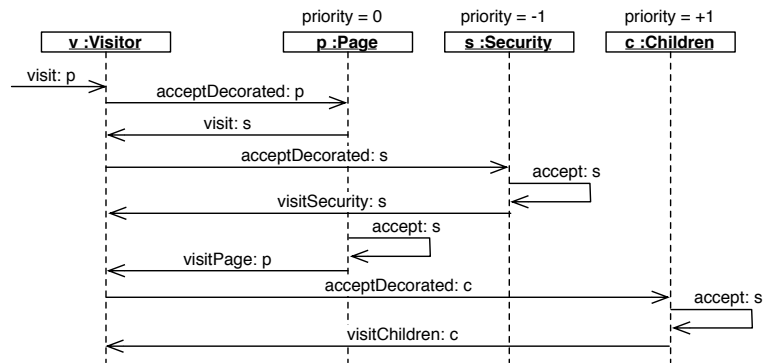


Figure 3.8: A Visitor interacting with a decorated page

Example. Figure 3.8 shows a sequence diagram of a Visitor operating on a page decorated with a security and a children decoration. Visiting the page `aVisitor visit: aPage`, triggers the callbacks for the decorations and

the page in the right order according to their priorities: `#visitSecurity:`, `#visitPage:` and `#visitChildren:`.

3.4.4 Context and Commands

Whenever Pier is browsed or edited it remembers within a context object the current state, this is the currently browsed structure, the running command, the user logged in, etc. Unlike most other content management systems Pier does not keep this information as a string in the URL or in session cookies. Seaside provides a nice abstraction over the low-level HTTP protocol and allows us to keep all the state within proper objects that persist along the session. It is therefore not necessary to manually serialize and de-serialize our context as strings.

Since every part of Pier can potentially modify the current context, say to navigate to a different structure, we must make sure that we do not lose the old context, since we might still need the original one for logging the changes with the persistency framework. With this in mind we decided to make context objects immutable. Sending a message that would modify the context does not touch the original object but instead returns a modified copy. It is the responsibility of the developer to make the new context the current one.

In Seaside the current context is hold in the top-level component of the wiki and can be requested or changed by raising a notification. We don't store the context within a global session object, since we would like to keep the possibility to embed Pier into existing Seaside applications that presumably already have their own session implementation. In the OmniBrowser view (see [Figure 3.5](#)), the current context is kept within the model of the browser.

Representing Actions as Commands

Every context references an associated command. Pier uses the Command design pattern [[Alpe98](#), page 245] to cleanly represent operations on the model using objects. Every command class is meta-described and therefore can be configured using an automatically built Magritte user interface, without having to know all the available commands that might have been loaded through extensions of the core framework. Commands are executed by the kernel, which gives transactional behavior for all modifications of the model. Since applied commands are logged to the history, we are able to provide multi-level undo facilities.

The core distribution of Pier comes with a small collection of command classes for basic actions on the model:

View. The view command plays a special role in the whole command hierarchy. It is the only command that does not modify the model, but represents a read-only view on the currently browsed structure. Its action semantics follows the Null Object Pattern [Wool96].

Edit. The edit command displays a Magritte form to edit the currently browsed structure. The editor is built using the descriptions of the structure that have been marked editable with the method `#beEditable`.

Add. The add command instantiates a new structure of the selected class, assigns the given name and adds it as a child of the currently browsed structure.

Remove. The remove command removes the currently browsed structure and all its children from the parent. The user is asked for confirmation before applying this command.

Copy. The copy command copies the currently browsed structure and all its children to a new location. References within the copied subtree are automatically updated.

Move. The move command moves the currently browsed structure and all its children to a new location. This command is also useful to rename structures. Other pages that reference the moved structure are automatically updated.

Sending the message `#execute` to a command instance executes this command under mutual exclusion, so that concurrent modifications of the domain model don't interfere with each other. In addition this ensures the modifications to be valid before processing and that they are logged in the persistency layer after execution.

Example. Adding a new structure to Pier is implemented as follows: `#do-Execute` is a hook method that is called from within the critical section in `#execute`. The first line actually adds the newly created child to the current page and remembers the child within a temporary variable `structure`. It then activates a new context by sending `#goto:command:` to enter the edit mode on the newly created child. However this new context won't be activated right away and it is remembered as the answer of the add command. In the meantime the Pier persistency framework is able to log the executed command together with the old context, so that it can be undone or replayed if necessary.

```
AddCommand>>doExecute
| structure |
self structure children
  add: (structure := self newChild).
self answer: (self context
  goto: structure
  command: structure editCommandClass new)
```

The command hierarchy offers a clean interface to modify the model of Pier. Actually every modification (or write access) to the model goes through a command, so that it can be logged and possibly undone at a later point in time. As we will see in [Section 3.6.2](#), having an initial state of the model and a list of logged commands with their associated contexts allows the implementation of a prevalence or changeset-like persistency mechanism, in which each change is stored with a time-stamp. Hence, it is not even necessary to keep the old versions of a page explicitly in the model, because they can be easily obtained by going back through the history of commands selecting all edit-commands on a particular page.

3.4.5 Environment

Pier unifies the look of the site with the wiki metaphor and allows one to define the look of the page using the wiki syntax itself. Thus people only have to learn one concept that can be used seamlessly in different areas. Anywhere within the system one is able to define a special page called *environment* that is invisible to the casual user and that defines the look of a portion of the application. An environment is shared between all the children of the same page, unless a new environment is defined that replaces the previous one. Since the environment is simply a page, it can be edited and modified like any other page.

The default environment page creating the standard look of Pier consists of the following piece of wiki text:

```
+Header+
+Views+ <br/> +Commands+ <br/> +Tree+ | +Contents+
Powered by Seaside and Pier
```

The first line with `+Header+` embeds a special header widget. Even though the *Header* structure could be yet another page, in this particular case we are using a Seaside component to draw and provide the necessary functionality. The next line creates a table containing the command and tree widget in a column on the left and the actual contents on the right. At the bottom we include a paragraph of static text that will be displayed on every page.

Furthermore, in Pier any Seaside component can be added exactly the same way as one would add a page into the wiki tree. In the above example we were

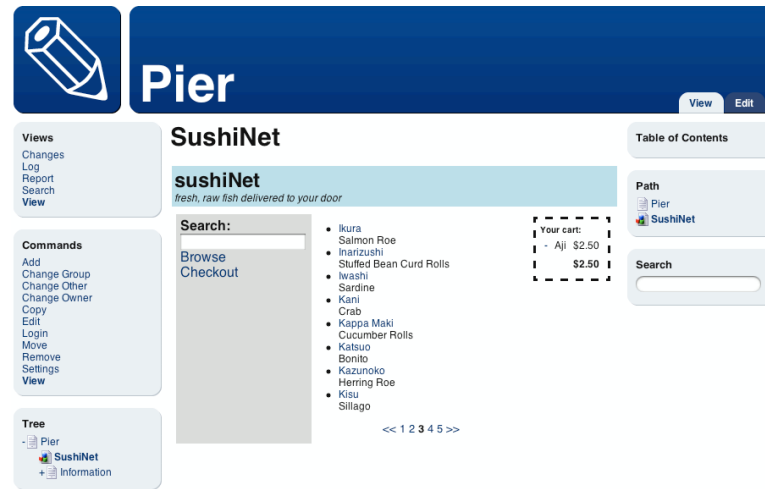


Figure 3.9: Seaside SushiNet application in Pier

using Seaside components that were particularly designed to be used within the wiki and provide its core functionality. Any other Seaside application, however, can be added exactly the same way. In Figure 3.9 one can see a “Sushi Web Shop” that is included with the Seaside framework and is often used to demonstrate the power of Seaside [Duca04]. Without changing a line of code in the embedded application and in Pier, the Seaside application can be plugged into the running content management system.

All embedded components and applications, no matter if they were especially designed for Pier or if they were originally used in a different context, can be configured through the Web interface. Thanks to the use of Magritte descriptions the settings of those components can be easily changed by the site administrator.

3.5 Extending Pier

As Pier has been designed to be extensible and customizable, we want to give some examples of small extensions in this section.

3.5.1 Fixing broken links

Since URLs and associated resources are changing from day to day it is a common issue that Web pages contain invalid links. There are plenty of tools available that address this problem by going through a Web site,

parsing the HTML and checking the validity of the links. In Pier we are able to address this issue simply by creating a subclass of `Visitor` and overriding the message `visitExternalLink:` to ask the link whether it is pointing to a valid resource and collect the broken ones within a collection. A user interface might then start this Visitor, display the broken links within a report and allow the responsible user to edit the links from one central place without being forced to go into every page and fix them manually. The only method to be implemented looks as follows:

```
BrokenLinkCollector>>visitExternalLink: anExternalLink
    anExternalLink isBroken
        ifTrue: [ collection add: anExternalLink ]
```

As we will demonstrate later on in [Section 3.6.1](#) we might also use the query engine and specify a query string like `kind = 'ExternalLink' AND isBroken = true` to achieve the same result.

3.5.2 Converting documents

It can be very convenient to convert a particular page or even a whole tree of pages to a different format than XHTML, for example for exporting or printing⁶. Since all the pages and documents are kept in one tree of objects it is trivial to write a Visitor that walks this tree of entities and exports the contents to formats like L^AT_EX, OASIS (Open Document Format for Office Application, OpenOffice) or RTF (Rich Text Format, Microsoft Word). In fact this is exactly the same way how the wiki syntax and the XHTML view for the Web browser are generated. The following code extract shows the part of the rendering Visitor that emits a L^AT_EX list:

```
LatexRenderer>>visitOrderedList: anOrderedList
    stream nextPutAll: '\begin{enumerate}'; cr.
    self visitAll: anOrderedList children.
    stream nextPutAll: '\end{enumerate}'; cr

LatexRenderer>>visitUnorderedList: aUnorderedList
    stream nextPutAll: '\begin{itemize}'; cr.
    self visitAll: aUnorderedList children.
    stream nextPutAll: '\end{itemize}'; cr

LatexRenderer>>visitListItem: aListItem
    stream nextPutAll: '\item '.
    self visitAll: aListItem children.
    stream cr
```

⁶In fact, the whole documentation of Magritte and Pier in the [Appendix](#) was automatically generated using this Visitor on the source code comments written using the Pier wiki syntax.

3.5.3 Security

Pier doesn't come with a built-in security framework, this means that out of the box there are no possibilities to restrict access to specific views or commands. However the implementation of Pier allows one to load a security system as a plug-in so that users can choose an implementation that suits their needs the best.

The key idea to enable that kind of pluggability is the use of decorations to attach security properties to any structure in the system. Visitors walking the model can then restrict access to a structure the user is not allowed to see. Details about the workings of Visitors can be found in [Section 3.4.3](#).

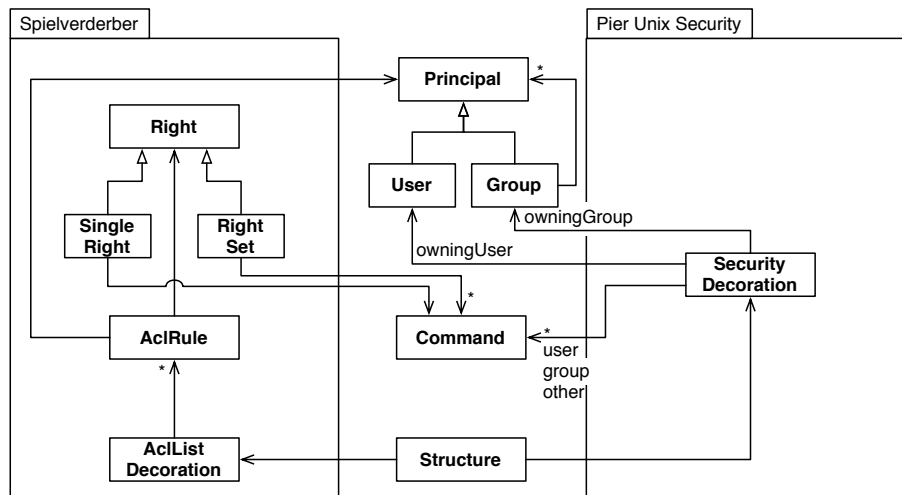


Figure 3.10: Two pluggable security architectures for Pier

Currently there are two security plug-ins available, as pictured in [Figure 3.10](#). Both implementations have a similar notion of users and groups, they define their access rules and their security decorations differently:

Access Control List. This security model [Plus05] is defined through access control lists (`AclList`), consisting of an ordered list of rules that control the permissions (`AclRule`). Each rule specifies three things: a user or group, a right, and whether that command is *allowed* or *denied*. One of the main advantages of this model is that permissions can not only be rejected but also explicitly granted. The drawback is the complexity: since the order of the rules is significant it can become difficult to manage sites with many large access control lists. The support of good tools is crucial here.

Unix Permissions. This package [Reng06] provides a lightweight implementation of a Unix-like security system. The security decoration has a notion of an owning *user* and an owning *group*, each with associated permissions. There is also a list of permissions for all *other* people, that neither match the owning user nor the owning group. Permissions are defined using the available command classes. This is basically the only difference to the security model in Unix where they merely have the permissions to read, write and execute. The package provides different commands that manage the permissions in similar ways as the well known Unix tools: `chown` to change the owning user, `chgrp` to change the owning group, and `chmod` to change the permissions.

3.6 Pier at the Meta-Level

Content management systems and wikis consist of a large number of input forms and dialogs that need to be built and validated manually. Developers need a way to specify how objects are structured and how they can be modified so that views and editors can be created almost automatically. In Pier, each domain element is described by a Magritte meta-description. Having such a description not only allows us to automatically create Seaside components as views for the Web, but also to build other user interfaces without having to write a single line of code. It automates searches on our domain model, implements persistency, etc.

Moreover, when changing the structure of a class one has to change the description at one single place and all the parts of the model and the user interface that rely on the provided descriptions immediately adapt to the new requirements, avoiding the need to refactor different parts of the code. Hence Magritte opens more possibilities to extend the system without having to patch existing code.

Meta-descriptions in Pier are not only used for describing the domain model itself but also for the Pier back-end object representation objects, such as the command objects mentioned in Section 3.4.4. As an example let's have a look at the copy command in Pier. On the class side there are two methods each returning a description. Both methods are initialized with a selector to access the value of the model; *i.e.*, for the title description only the read-accessor `title` is specified, but Magritte will automatically define the write-accessor `#title:`. The descriptions of the title and target are tagged to be required, which means that neither fields can be left empty.

```
CopyCommand class>>descriptionTitle
  ^ (StringDescription selector: #title label: 'Title' priority: 100)
    beRequired;
    yourself
```



```
CopyCommand class>>descriptionTarget
  ^ (StructureDescription selector: #target label: 'Target' priority: 200)
    beRequired;
    yourself
```

When asking an instance of such a copy command for its description, Magritte collects all the methods on the class side starting with the name `description` and returns a composed description consisting of the two elements as seen above. The value of the priority is used to sort the descriptions as preferred to give a consistent look in the user interface.

There are multiple uses of such meta-descriptions. The most immediate one is that a description is used to create a visual Seaside component. Getting a Seaside component allowing the user to edit the command instance is as simple as sending `#asComponent`. Usually the returned component is then decorated with a form, displaying a save and a cancel button, and a validator, catching and displaying validation errors. In a very similar way we are able to send `#asMorphic` to the same command to get a dialog for the OmniBrowser user interface.

In the following sections we discuss the implementation of the search engine and the persistency framework with the help of Magritte.

3.6.1 Searching

Content management systems tend to grow over time, hence it becomes very important to have sophisticated ways to locate the desired information. In most cases we want to search for a page containing a particular substring, however sometimes it would be more precise to only look for pages that satisfy a certain condition. Pier with the help of the meta-descriptions of Magritte allows one to write such conditions in the search field and display the matching pages. The query `kind = 'Table' AND rowCount > 3` returns all the pages with tables that have more than 3 rows and `url matches: '*.ch/*'` returns pages with external links having a swiss domain.

To implement this functionality we have written a parser that reads the search expression and builds a tree of conditions, see [Section 2.6.3](#). When we send the `#value:` to the root node of this condition tree, we either get `true` if the argument matches the criterion or `false` if it doesn't match as a return value. The condition tree is evaluated recursively using an escaper to abort the evaluation immediately if nothing can change the result of the expression anymore, which significantly improves the speed of query processing. To determine if a certain basic condition such as `title beginsWith: 'Pier'` is met, the meta-descriptions come into play again: the model object to be

checked is asked for its descriptions and it is checked if it has got an attribute called `title` and if this attribute is usable with the condition `beginsWith:`. If those two preconditions are fulfilled the value is read from the domain model, the comparison is done and the result is returned. Again we have a Visitor that walks over the wiki tree and collects all the possible matches. A simple widget is used to display the result of the query.

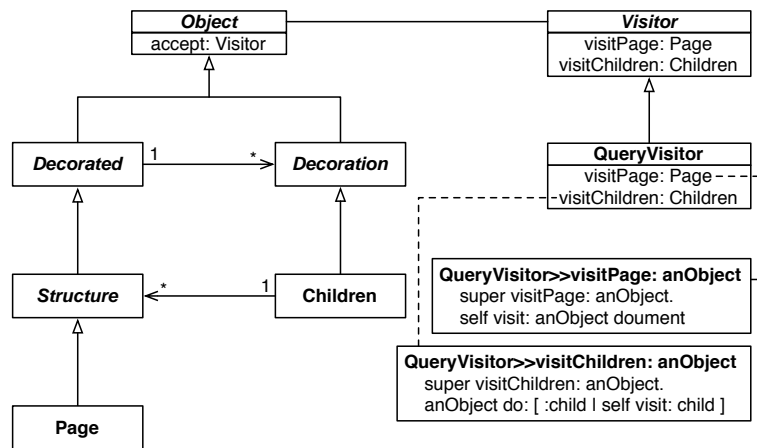


Figure 3.11: Walking through the Pier model using a Visitor

The following example shows how the query string is parsed and passed into the Visitor, which descends into each wiki structure, as seen in Figure 3.11, and then collects and returns all the matching structures.

```

SearchWidget>>search
  searchResults := QueryVisitor
    start: self context kernel root
    query: (RelationParser parse: self queryString)

BasicCondition>>value: anObject
  | description |
  description := anObject description
  at: self selector
  ifAbsent: [ ^ false ].
  ^ super value: (anObject readUsing: description)

```

3.6.2 Persistency and Versioning

Pier takes a prevalence [Wues] or change-set-based approach to version its data. The idea is to keep the whole data in RAM – if there isn't enough RAM on the server it will be transparently swapped out by the operation system – so the system runs very fast as no objects have to be de-serialized. To avoid losing data, every night, or at any reasonable time, a snapshot of

the whole page tree is saved. In addition all commands that are executed on the model are serialized immediately after being processed. The meta-descriptions of the command tells the persistency layer how the object has to be serialized and eventually restored later on. During crash recovery, Pier retrieves its last saved state from the snapshot and then reads in the commands and applies them to the model exactly as if it had just come from the user interactions.

With this approach we get versioning and undo facilities for free. Suppose we want to see all the changes that have been made to a specified page, we just have to go through the command log and select all the edit commands of this particular page. Loading them allows us to see the changes of that page, and restore any old version by re-applying the command.

3.6.3 Adaptive Forms

Often end user of a content management system not only want to collect data in the form of text documents and files but also in more structured ways, *e.g.*, addresses within an address book, movies in their DVD collection, items in a to-do list, etc. A possible use case of an actor database can be seen in [Figure 3.12](#). With the help of Magritte it is easy to enable adaptive forms, see [Section 2.4.1](#), that can be configured by end users. This section will briefly document the code that was written to enable such a functionality.

We will start by defining a new structure class that behaves like a normal page and displays the contents of the form as a read-only view (see [Figure 3.12.c](#)). Then we will enable the edit command to take the custom fields of the new form structure into account (see [Figure 3.12.b](#)). Finally we will provide a meta-edit command to change the meta-model itself (see [Figure 3.12.a](#)). Noticeably the implementation only requires to add two new classes, has no view-specific code and therefore works out of the box for all available views.

First we define a new structure (see [Section 3.4.2](#)) by subclassing `Structure`, having two instance variables, accessors and an initialize method. We also implement the method `#postCopy` to copy the model but not the meta-model.

```
Structure subclass: #Form
    instanceVariableNames: 'model metamodel'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Pier-Forms'

Form class>>isAbstract
    ^ false
```

(a) Meta Editor

Edit Form

Kind: Container
 Label: Actor
 Priority: 0
 Readonly: ☐ Readonly
 Visible: ☒ Visible
 Required: ☐ Required
 Persistent: ☒ Persistent

Elements:	Kind	Label	Priority	Readonly	Visible	Required	Persistent	
Memo	Biography	0	false	true	false	true		Edit Remove
File	Photo	0	false	true	false	true		Edit Remove
Date	Birthday	0	false	true	false	true		Edit Remove
Number	Height	0	false	true	false	true		Edit Remove

Boolean ☐ Add


Save Cancel

(b) Editor

Edit

Title: Scarlett Johansson

Biography: Johansson's performance as Grace in The Horse Whisperer (1998) earned her a Hollywood Reporter Young Star award. And previously her Manny & Lo (1996) role received a nod by the Independent Spirit Awards.

Photo:  Remove

Birthday: 22 November 1984 Choose

Height: 1.63

Save Cancel

(c) View

Scarlett Johansson

Biography Johansson's performance as Grace in The Horse Whisperer (1998) earned her a Hollywood Reporter Young Star award. And previously her Manny & Lo (1996) role received a nod by the Independent Spirit Awards. Scarlett has an older brother and sister and a twin brother. She divides her time between New York with her dad, and Los Angeles with her mother. Her acting career was launched in the off-Broadway production of 'Sophistry' with Ethan Hawke.

Photo 

Birthday 22 November 1984

Height 1.63

Figure 3.12: (a) Defining an adaptive form for a movie actor database. (b) Editing an instance of the adaptive form. (c) Presenting a read-only view of the instance of the adaptive form.

```
Form>>metamodel
  "Answer a Magritte container with the description of the receiver."

  ^ metamodel

Form>>metamodel: aDescription
  metamodel := aDescription

Form>>model
  "Answer a dictionary mapping the descriptions of the receiver to
  actual values."

  ^ model

Form>>model: aDictionary
  model := aDictionary

Form>>initialize
  super initialize.
  self model: Dictionary new.
  self metamodel: Container new
```

```
Form>>postCopy
  super postCopy.
  self model: self model copy
```

Next we override the method `#description` to return a concatenation of the description of the super class and the adaptive meta-model of the receiver. To tell Pier that it should use our adaptive descriptions to build the default editor, we have to make it editable by sending the message `#beEditable`.

```
Form>>description
  ^ (Container withAll: super description) ,
    (self metamodel do: [ :each | each beEditable ]
```

To be able to read the data from the two possible sources, the receiver as described by the superclass or our dictionary as described by our adaptive meta-model, we have to override the methods `#readUsing:` and `#write:-using:`. These two methods are invoked by Magritte to read and write values from a described object.

```
Form>>readUsing: aDescription
  "Answer the actual value described by aDescription. If our meta-model
  includes aDescription return the associated value from the dictionary,
  else use the super implementation."

  ^ (self metamodel includes: aDescription)
    ifTrue: [
      self model
        at: aDescription
        ifAbsent: [ aDescription default ] ]
    ifFalse: [ super readUsing: aDescription ]

Form>>write: anObject using: aDescription
  "Set the value described by aDescription to be anObject. If our meta-
  model includes aDescription put anObject into the dictionary, else use
  the super implementation."

  (self metamodel includes: aDescription)
    ifTrue: [ self model at: aDescription put: anObject ]
    ifFalse: [ super write: anObject using: aDescription ]
```

To get the read-only view of the new structure class working we need to define a method `#document` returning a Pier document composite. We programmatically build a list by iterating over the meta-model and transforming our model to strings:

```
Form>>document
  ^ Document new
    add: (UnorderedList new
      add: (self metamodel children collect: [ :desc |
```

```

        ListItem
        with: (Text with: '<b>' , desc label , '</b>');
        with: (Text with: (desc
            toString: (self readUsing: desc)));
        yourself ]);
    yourself);
    yourself

```

The only missing piece is the ability to edit the meta-model. We do this by implementing a new command class:

```

Command subclass: #EditFormCommand
    instanceVariableNames: 'metamodel'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Pier-Forms'

```

```

EditFormCommand class>>label
    ^ 'Edit Form'

```

```

EditFormCommand class>>structureClass
    ^ Form

```

```

EditFormCommand class>>isAbstract
    ^ false

```

Then we implement accessors and override the method `#description` to return the description of the meta-model:

```

EditFormCommand>>metamodel
    "Answer the meta-model of the command, fetch it from the current
    structure if it is not initialized."

    ^ metamodel ifNil: [ metamodel := self structure metamodel ]

EditFormCommand>>metamodel: aDescription
    metamodel := aDescription

EditFormCommand>>description
    ^ self metamodel description

```

Last but not least we have to implement the method `#doExecute` to actually change our model:

```

EditFormCommand>>doExecute
    super doExecute.
    self structure metamodel: self metamodel

```

This is all needed to get a working implementation as shown in [Figure 3.12](#). Still there is some space left for improvements, such as to provide better views and user interfaces to make it simpler to utilize.

3.7 Lessons Learned

During the implementation of Pier and its ancestor SmallWiki, we learned that having a fully object-oriented design is the key to efficiently implement an extensible content management system that adapts to a wide variety of needs. In the following paragraphs we compare some implementation details of SmallWiki, Pier and other wiki implementations:

Testing. Compared with SmallWiki, Magritte and Pier increased the number of unit tests from 200 to more than 2500, covering the whole model of both frameworks. This makes it possible to change and verify the code and is extremely useful when porting Pier to other Smalltalk dialects or when writing extensions that could break existing code.

Parser. Using a parser to read the Wiki input, to build a proper object model and to walk through it using Visitors saves a lot of code: the current implementation of Pier featuring scanner, parser and document hierarchy consists only of 550 lines of Smalltalk code, whereas the same functionality implemented for Wikipedia [WikiPedi] using regular-expressions requires more than 3000 lines of code (excluding comments). Moreover these regular-expressions are duplicated throughout the code base of Wikipedia, for example to implement the query engine, which makes it extremely difficult to change and enhance the syntax.

Structures. In the first version of SmallWiki, we distinguished between a folder (this is a page with children) and a page. This led to problems because it was difficult to change the structure of a Wiki after the fact. In the new version, we only have pages and no folders but any structure can be decorated to get children. Hence any structure can play the role of a folder. A page can also lose its children. This means the user is not forced to decide up front how his Web site will be structured, but is able to add and remove children later on as desired. Pier also provides an interface to move and copy whole subtrees to different locations easily.

Separation. SmallWiki was designed to be used within a Web context [Reng03]. It was built on top of its own Web framework. However, Web application development is difficult when having to deal with the shortcomings of the HTTP protocol as the right abstractions are missing [Duca04]. In SmallWiki the model and the view were strongly coupled. For example an action to be performed on a page was a mixture between a Command design pattern [Gamm95] and the associated Web view. It was then nearly impossible to use a command in a different view. Now Pier cleanly separates the model and the view

in different packages that can be loaded and used independently.

View. In SmallWiki all the application state was kept as strings in the URL, in its query parameters, in HTTP header fields and in associated session cookies, exactly the way most of today's Web applications do. Using Seaside as a default view allows us to introduce a much cleaner solution. Seaside provides a nice abstraction over this low-level protocol and we are now able to keep all our state within the application components themselves as proper objects. It is therefore not necessary to manually serialize and de-serialize our objects as strings.

Embedding. Structures can be embedded into each other by creating a special kind of reference. This greatly enhances the possibilities to layout and structure the Wiki. Pier supports absolute and relative links, so that editors can easily create navigation facilities between the nested structures.

Commands. Modifying the model through the use of a clean implementation of the command pattern enables the implementation of a prevalence-like framework. It logs the history of commands to allow one to restore any point in the past by loading a snapshot and subsequently applying the stored commands. Furthermore having the whole command history available gives us the possibility to undo and redo modifications.

Meta-Description. We learned that having a powerful meta-model brings a lot of flexibility to different areas of the framework. Without writing additional code we are able to alter different parts of Pier, such as the views, the search engine and the persistency solely by changing or adding Magritte meta-descriptions.

Persistency. Persistency and versioning is a crucial part of any Wiki. In SmallWiki we were using a simple snapshot mechanism, dumping out all the structures in user-defined intervals. The obvious problem here is that if the computer crashes just before doing a snapshot all the changes since the last snapshot are lost. The versioning of the pages was achieved by keeping a collection of all the old pages within the model, which has disadvantages as well: old versions are only accessed rarely and therefore it is not efficient to keep them in memory all the time. In addition, the memory footprint of SmallWiki never shrank, since all the changes had to be versioned, therefore even deleting parts of the Wiki didn't reduce its actual size. This could lead to performance problems when saving and loading a snapshot of a huge Wiki with lots of mutations over the time. Pier provides a prevalence-based approach to versioning, so that every change is stored to the filesystem and only the current model is kept in memory. Any point after a

snapshot can be easily restored by reapplying the commands from the history.

3.8 Summary

Content management systems and Wikis both offer quick and efficient ways to collaborate via simple Web browser interfaces. However, as Web sites grow, more advanced functionalities (such as advanced management, maintenance and search operations) need to be incorporated. Current implementations based on string manipulation are poorly suited to support this new generation of collaboratively created Web sites.

In this chapter we have presented Pier, a fully object-oriented content management system that is described using Magritte as a meta-model – forming the conceptual backbone of the implementation – and that uses the Seaside framework to overcome traditional HTTP limitations. Seaside lets Pier cleanly divide the application domain model from the UI, alleviating the need for object serialization. The resulting combination was shown to be very customizable.

Our long term goal for Pier is to define an environment to enable user-scriptable Web applications, similar to HyperCard in its time.

Chapter 4

Conclusion

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

— Christopher Alexander

As we have observed while developing several real world applications, having a meta-framework such as Magritte greatly reduces recurrent work. Often it is much simpler to write a generic interpreter of the meta-model, than to manually build specific implementations of the functionality in different places of the application. Hence, the use of Magritte not only reduces the overall code size, but it makes the application more adaptable to changing needs of customers and reduces the number of possible bugs: developers only change the description at one single place in the source code, without having to refactor all places that deal with the object. In addition, complex meta-model manipulations are made once and for all by a meta-programmer, and the other programmers and users can benefit from them.

	Model Code	Web-View	Morphic-View
Magritte	2490	1801	286
Pier	2978	1021	249
Pier Security	472	0	0
Pier Forms	61	0	0

Table 4.1: Lines of Code in Magritte and Pier.

Table 4.1 shows the lines of code (LOC) of Magritte, Pier and some extensions thereof. The columns show the difference between the model code and the code used to generate the different views. It is interesting to see that the security and the form extension don't have any view-specific code, since their user interface is all automatically generated using Magritte. There is only very little model code that makes up the *Pier Forms* package – an extension we presented in Section 3.6.3 allowing end users to define their own forms in the system – because most of this functionality is already provided by Magritte itself.

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 stated by Ralph Johnson [Yode02] a meta-model 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 meta-model. We have not observed any performance cost or noticeable speed reductions through the use of Magritte. Other factors such as the network connection or the persistency back-end are by far more critical for business applications than the use of an underlying meta-model.

4.1 Related Work

Yoder *et al* propose the type-square design pattern [Yode01], based on the type object that separates the entity from its entity type [John98]. Magritte uses these patterns as well, but it makes some generalizations, as seen in Figure 4.1: 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. A couple of descriptions refer to other descriptions, such as the *container descriptions*, the *option descriptions* and the *relationship descriptions*. *Strategies* and *Business Rules* are modeled in Magritte using *conditions* and different *Visitors* that can be associated to any description object and that are meta-described, so that they can be changed or customized by end users as well.

Formulator [Formulat] and Mewa [Lien03] propose frameworks that ease the

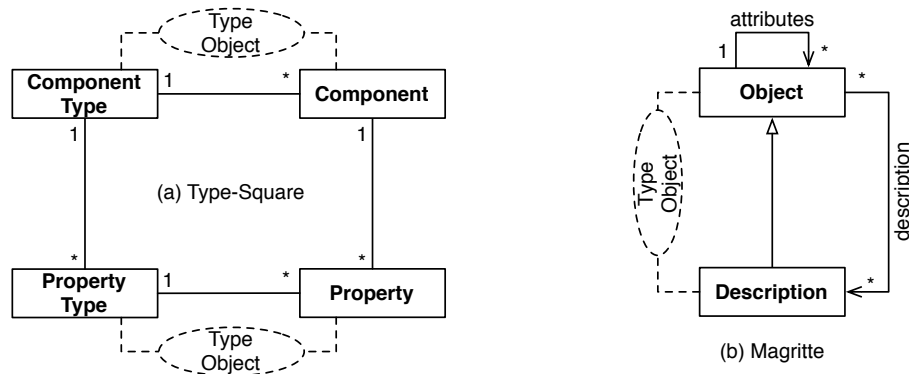


Figure 4.1: (a) The type-square, and (b) the meta-recursive model of Magritte are both making extensive use of the type-object design pattern.

creation and validation of Web forms, however neither approach uses the meta-model for anything other than building forms. Formulator includes basic Web interfaces to build forms interactively. Mewa requires the developer to write all the code to generate a meta-model. Both frameworks have difficulties with customizing parts of the automatic form generation.

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 adaptive models that are not able to describe themselves. Therefore they require a lot of additional work 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 [Mull05b] present an approach to platform-independent Web application modeling and development in the context of model-driven engineering. A specific meta-model (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 all over these models to write methods and actions, specify constraints and express conditions.

We decided against using yet another language and preferred to use the complete power of our chosen development language Smalltalk, as it offers a simple object-oriented model with OCL-like iterations and a powerful set of development tools (hot debugger, session-debugging, hot-server recompilation). In addition, Seaside, the framework we use to develop Web applications, allows us to build Web applications as if they were desktop

applications. Therefore if we were to adopt a generative approach we would lose the power of Seaside [Seaside, Duca04].

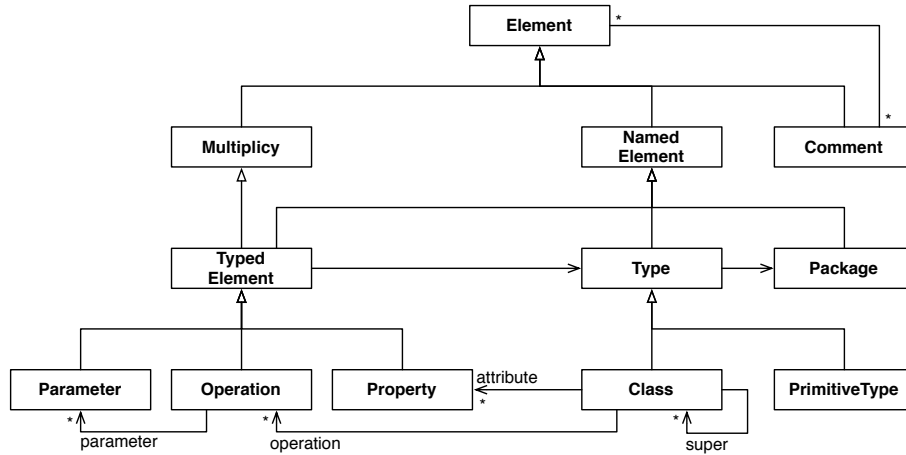


Figure 4.2: The Essential Meta-Object Facility (EMOF)

EMOF. Figure 4.2 shows a class diagram of the Essential Meta-Object Facility (EMOF), a standard for model driven engineering defined by the Object Management Group (OMG) [Grou04]. Contrary to EMOF Magritte is not designed as a four-layered architecture. In Magritte there is not distinction drawn between the meta-meta-model (M3), the meta-model (M2), the model (M1) and the instances (M0). Descriptions are objects that can all be seen at different levels at once: the meta-level, the model-level, etc. In Smalltalk everything is an object, in particular descriptions and classes are objects that can be described as well.

There is no notion of named elements, packages or namespaces in Magritte. Descriptions don't need to be identified by name, Factory Methods [Alpe98, page 63] are used to build description instances and from then on they are merely stored in instance variables and passed around using object references. Descriptions are not part of the underlying source code and don't necessarily describe something that has been written by an application developer, hence they neither need a name nor belong to a package or namespace. Descriptions are compared using object identity, consequently two descriptions are identical if they are represented by the same object. Descriptions happily live together with the other model-instances in the memory or in external data sources.

Compared to MOF Magritte has no notion of instantiation, inheritance and classes. We describe objects that are already 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 every object can have multiple descriptions that can be used in sequence or in parallel on the same object. Additionally descriptions can be shared among different objects, as long as the object implements the described elements.

The Magritte description classes define a type hierarchy for the Smalltalk class library. This is similar to the subclasses of **Type** in EMOF, where a distinction between classes and primitive types is made. An instantiated Magritte description can be seen as the EMOF class **Property**. Magritte also provides means to define multiplicity, but not as fine grained as this is done for MOF. In Magritte there is no built-in functionality to describe behavioral aspects, such as operations, their parameters and return values. The pluggable design of Magritte however allows one to use descriptions to model simple operations on objects. This has been shown in [Section 2.5](#) where we provided different comparators with user definable arguments. It is desired to extend this model to allow describing arbitrary operations on objects. Magritte supports condition objects on its descriptions, this is similar to the constraints that are part of the Complete Meta-Object Facility (CMOF) only.

WebML. WebML [[Ceri00](#)] 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 meta-model changes, all the users of the meta-model behave in the new way automatically.

4.2 Lessons Learned

- Meta-descriptions considerably enhance the possibilities to refactor and change existing code, since the changes are localized to one place. In other words “Describe once, get everywhere”.
- Smalltalk as a meta-language offers the possibility to use all the tools provided by the development environment: browser, debugger, ver-

sioning, testing, refactoring, etc. Moreover it eases the entry level as developers do not need to learn another language.

- Using the same descriptive paradigm on different meta-levels turns Magritte into a powerful recursive framework: being able to describe any kind of domain-objects, including descriptions themselves makes it possible to enable end user customizability.
- The simplicity and extensibility of Magritte makes it possible to write interpreters that perform completely different actions on described models, as the ones provided with the framework (building Web editors, serializing domain-models, querying data).

4.3 Further Work

We have described our practical knowledge of using a meta-model integrated in the reflective meta-model of Smalltalk to support the development of flexible Web applications. Our meta-model is self-described, which enables end user customization. We demonstrate that we could reap the benefits of the two worlds: on the one hand we keep our efficient and dynamic object-oriented programming with an outstanding tooling context, and at the same time we gained the flexibility and compactness of meta-descriptions to factor out repetitive tasks of our application development.

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 that can be modified by the user on the fly. Also we would like to implement ideas from EMOF to model operations on objects.

Documentation

Magritte-Model-Core

MACompatibility

Superclasses `Object`

I am providing all the platform compatibility code on my class side, so that porting to different Smalltalk dialects can concentrate in a single place.

MADistribution

Superclasses `Object`

I am responsible for building a distribution and publishing a complete package on SqueakMap. All my settings, such as which packages I depend on, are defined on the class side and can be overridden to create different distributions based on Magritte.

building

- `dump`
Dump the SAR archive to the file-system.
- `publish`
Publish the package as a new release on SqueakMap.

MAObject

Superclasses `Object`

Subclasses `MAAccessor`, `MAAllCondition`, `MAAnyCondition`, `MAAutoSelectorAccessor`, `MABlockAccessor`, `MABooleanDescription`, `MACachedMemento`, `MAChainAccessor`, `MACheckedMemento`, `MAClassDescription`, `MAColorDescription`, `MACondition`, `MAContainer`, `MAContainerAccessor`, `MADateDescription`, `MADescription`, `MADictionaryAccessor`, `MADurationDescription`, `MAElementDescription`, `MAFalseCondition`, `MAFileDescription`, `MAMagnitudeDescription`, `MAMemento`, `MAMemoDescription`, `MAMultipleOptionDescription`, `MANoneCondition`, `MANullAccessor`, `MANumberDescription`, `MAOptionDescription`, `MAPasswordDescription`, `MAPriorityContainer`, `MAReferenceDescription`, `MARelationDescription`, `MASelectorAccessor`, `MASelectorCondition`, `MASingleOptionDescription`, `MAStraitMemento`, `MAStraintMemento`, `MAStraintMemento`, `MASymbolDescription`, `MATableDescription`, `MATimeDescription`, `MATimeStampDescription`, `MATokenDescription`, `MAToManyRelationDescription`, `MAToOneRelationDescription`, `MATrueCondition`, `MAVariableAccessor`

I provide functionality available to all Magritte objects. I implement a dictionary of properties, so that extensions can easily store additional data.

accessing

- `properties`
Answer the property dictionary of the receiver.
- `propertyAt: aKey`
Answer the value of the property `aKey`, raises an error if the property doesn't exist.
- `propertyAt: aKey ifAbsentPut: aBlock`
Answer the value of the property `aKey`, or if the property doesn't exist adds and answers the result of evaluating `aBlock`.
- `propertyAt: aKey ifAbsent: aBlock`
Answer the value of the property `aKey`, or the result of `aBlock` if the property doesn't exist.
- `propertyAt: aKey put: aValue`
Adds or replaces the property `aKey` with `aValue`.

comparing

- `hash`
Answer a `SmallInteger` whose value is related to the receiver's identity. Also redefine the message `#=` when redefining this message.

- `= anObject`

Answer whether the receiver and the argument represent the same object. This default implementation checks if the species of the compared objects are the same, so that superclasses might call `super` before performing their own check. Also redefine the message `#hash` when redefining this message.

copying

- `postCopy`

This method is called whenever a shallow copy of the receiver is made. Redefine this method in subclasses to copy other fields as necessary. Never forget to call `super`, else class invariants might be violated.

testing

- `hasProperty: aKey`

Test if the property `aKey` is defined within the receiver.

Magritte-Model-Models

MAAdaptiveModel

Superclasses `Object`

I am an adaptive model referencing a dynamic description of myself and a dictionary mapping those descriptions to actual values.

accessing

- `description`

Answer the description of the receiver.

- `values`

Answer a dictionary mapping description to actual values.

model

- `readUsing: aDescription`

Answer the actual value of `aDescription` within the receiver, `nil` if not present.

- **write: anObject using: aDescription**
Set **anObject** to be that actual value of the receiver for **aDescription**.

MAFileModel

Superclasses `Object`

I represent a file with filename, mimetype and contents within the Magritte framework.

querying

- **extension**
Answer the file-extension.
- **maintype**
Answer the first part of the mime-type.
- **size**
Answer the size of the file.
- **subtype**
Answer the second part of the mime-type.

testing

- **isApplication**
Return **true** if the mimetype of the receiver is application-data. This message will match types like: application/postscript, application/zip, application/pdf, etc.
- **isAudio**
Return **true** if the mimetype of the receiver is audio-data. This message will match types like: audio/basic, audio/tone, audio/mpeg, etc.
- **isImage**
Return **true** if the mimetype of the receiver is image-data. This message will match types like: image/jpeg, image/gif, image/png, image/tiff, etc.
- **isText**
Return **true** if the mimetype of the receiver is text-data. This message will match types like: text/plain, text/html, text/sgml, text/css, text/xml, text/richtext, etc.

- `isVideo`
Return `true` if the mimetype of the receiver is video-data. This message will match types like: `video/mpeg`, `video/quicktime`, etc.

MATableModel

Superclasses `Object`

I am a model class representing a table within the Magritte framework. Internally I store my cells within a flat array, however users may access data giving *row* and *column* coordinates with `#at:at:` and `#at:at:put:`. I can support reshaping myself, but of course this might lead to loss of data-cells.

accessing

- `at: aRowIndex at: aColumnIndex`
Answer the contents of `aRowIndex` and `aColumnIndex`. Raises an error if the coordinates are out of bounds.
- `at: aRowIndex at: aColumnIndex put: aValue`
Set the contents of `aRowIndex` and `aColumnIndex` to `aValue`. Raises an error if the coordinates are out of bounds.
- `columnCount`
Answer the column count of the table.
- `rowCount`
Answer the row count of the table.

operations

- `reshapeRows: aRowCount columns: aColumnCount`
Change the size of the receiving table to `aRowCount` times `aColumnCount`, throwing away elements that are cut off and initializing empty cells with `nil`.

Magritte-Model-Description

MABooleanDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElement-`

Description

I am a description of the Boolean values `true` and `false`. My visual representation could be a check-box.

MAClassDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`

I am a description of Smalltalk classes, possible values can be any of `Smalltalk allClasses`.

MAColorDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`

I am a description of colors, possible values are instances of `Color`. My visual representation could be a color-chooser.

MAContainer

Superclasses `Object`, `MAObject`, `MADescription`

Subclasses `MAPriorityContainer`

I am a container holding a collection of descriptions, all instances of subclasses of `MAElementDescription`. I keep my children within an `OrderedCollection`, but I don't sort them according to their priority.

I fully support the collection protocol: descriptions can be added and removed. Moreover I implement most enumeration methods, so that users are able to iterate (`do:`), filter (`select:`, `reject:`), transform (`collect:`), extract (`detect:`, `detect:ifNone:`), and test (`allSatisfy:`, `anySatisfy:`, `noneSatisfy:`) my elements.

MADateDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAMagnitudeDescription`

I am a description of dates, possible values are instances of `Date`. My visual representation could be a date-picker.

MADescription

Superclasses `Object`, `MAObject`

Subclasses `MABooleanDescription`, `MAClassDescription`, `MAColorDescription`, `MAContainer`, `MADateDescription`, `MADurationDescription`, `MAElementDescription`, `MAFileDescription`, `MAMagnitudeDescription`, `MAMemoDescription`, `MAMultipleOptionDescription`, `MANumberDescription`, `MAOptionDescription`, `MAPasswordDescription`, `MAPriorityContainer`, `MARefereceDescription`, `MARelationDescription`, `MASingleOptionDescription`, `MAStrngDescription`, `MASymbolDescription`, `MATableDescription`, `MATimeDescription`, `MATimeStampDescription`, `MATokenDescription`, `MAToManyRelationDescription`, `MAToOneRelationDescription`

I am the root of the description hierarchy in Magritte and I provide most of the basic properties available to all descriptions. If you would like to annotate your model with a description have a look at the different subclasses of myself.

Example

If your model has an instance variable called `title` that should be used to store the title of the object, you could add the following description to your class:

```
MyModel class>>descriptionTitle
  ^ (MAStringDescription auto: #title label: 'Title')
    beRequired;
    yourself.
```

The selector `#title` is the name of the accessor method used by Magritte to retrieve the value from the model. In the above case Magritte creates the accessor method and the instance variable automatically, if necessary. The label is used to give the field a name and will be printed next to the input box if a visual GUI is created from this description.

The write-accessor is automatically deduced by adding a colon to the read-selector, in this example `#title:`. You can specify your own accessor strategy using one of the subclasses of `MAAccessor`. If you have multiple description within the same object, the priority field is used to order them. Assign a low priority to have descriptions printed first.

accessing

- **accessor**
Answer the access-strategy of the model-value described by the receiver.

accessing-configuration

- **kind**
Answer the base-class (type) the receiver is describing. The default implementation answers the most generic class: `Object`, the root of the Smalltalk class hierarchy. Subclasses might refine this choice.
- **name**
Answer the name of the description, a human-readable string describing the type.

accessing-properties

- **comment**
Answer a comment or help-text giving a hint what this description is used for. GUIs that are built from this description might display it as a tool-tip.
- **conditions**
Answer a collection of additional conditions that need to be fulfilled so that the described model is valid. Internally the collection associates conditions, that are either blocks or subclasses of `MACondition`, with an error string.
- **label**
Answer the label of the receiving description. The label is mostly used as an identifier that is printed next to the input field when building a GUI from the receiver.
- **persistent**
Answer `true` if the model described by the receiver is persistent.
- **priority**
Answer a number that is the priority of the receiving description. Priorities are used to give descriptions an explicit order by sorting them according to this number.
- **readonly**
Answer `true` if the model described by the receiver is read-only.

- **required**
Answer **true** if the model described by the receiver is required, this is it cannot be **nil**.
- **visible**
Answer **true** if the model described by the receiver is visible, as an opposite to hidden.

accessing-strings

- **stringReader**
Answer a Visitor that can be used to parse the model described by the receiver from a string.
- **stringWriter**
Answer a Visitor that can be used to convert the model described by the receiver to a string.
- **undefined**
Answer a string that is printed whenever the model described by the receiver is **nil**.

converting

- **asContainer**
Answer a description container of the receiver.

operators

- **, aDescription**
Concatenate the receiver and **aDescription** to one composed description. Answer a description container containing both descriptions.
- **<= anObject**
Answer whether the receiver should precede **anObject** in a priority container.

strings

- **fromStringCollection: aCollection**
Answer a collection of objects being parsed from **aCollection** of strings.

- `fromStringCollection: aCollection reader: aParser`
Answer a collection of objects being parsed from `aCollection` of strings using `aParser`.
- `fromString: aString`
Answer an object being parsed from `aString`.
- `fromString: aString reader: aParser`
Answer an object being parsed from `aString` using `aParser`.
- `toStringCollection: aCollection`
Answer a collection of strings being formatted from `aCollection`.
- `toStringCollection: aCollection writer: aFormatter`
Answer a collection of strings being formatted from `aCollection` using `aFormatter`.
- `toString: anObject`
Answer a string being formatted from `anObject`.
- `toString: anObject writer: aFormatter`
Answer a string being formatted from `anObject` using `aFormatter`.

testing

- `hasChildren`
Answer `true` if the receiver has any child-descriptions. A description container usually has children.
- `hasComment`
Answer `true` if the the receiver has got a non empty comment.
- `hasLabel`
Answer `true` if the the receiver has got a non empty label.
- `isContainer`
Answer `true` if the receiver is a description container.
- `isDescription`
Answer `true` if the receiver is a description.

validation

- `addCondition: aCondition labelled: aString`
Add `aCondition` as an additional validation condition to the receiver

and give it the label `aString`. The first argument is either a block-context, a composite of the subclasses of `MACondition`, or any other object that responds to `#value:` with `true` or `false`.

- `isSatisfiedBy: anObject`
Answer `true` if `anObject` is a valid instance of the receiver's description.
- `validateConditions: anObject`
Validate `anObject` to satisfy all its custom conditions.
- `validateKind: anObject`
Validate `anObject` to be of the right kind.
- `validateRequired: anObject`
Validate `anObject` not to be `nil` if it is required.
- `validateSpecific: anObject`
Validate `anObject` to satisfy all its description specific validation rules. Subclasses mostly want to override this method.
- `validate: anObject`
Validate `anObject` in the context of the describing-receiver, raises an error in case of a problem. If `anObject` is `nil` and not required, most tests will be skipped. Do not override this message, instead have a look at `#validateSpecific:` what is usually a better place to define the behaviour your description requires.

MADurationDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAMagnitudeDescription`

I am a description of durations, possible values are instances of `Duration`.

MAElementDescription

Superclasses `Object`, `MAObject`, `MADescription`

Subclasses `MABooleanDescription`, `MAClassDescription`, `MAColorDescription`, `MADateDescription`, `MADurationDescription`, `MAFileDescription`, `MAMagnitudeDescription`, `MAMemoDescription`, `MAMultipleOptionDescription`, `MANumberDescription`, `MAOptionDescription`, `MAPasswordDescription`, `MAReferenceDescription`,

`MARelationDescription`, `MASingleOptionDescription`, `MAStringDescription`, `MASymbolDescription`, `MATableDescription`, `MATimeDescription`, `MATimeStampDescription`, `MATokenDescription`, `MAToManyRelationDescription`, `MAToOneRelationDescription`

I am an abstract description for all basic description types.

MAFileDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`

I am a description of files, their contents, filename and mime-type. Possible values include instances of `MAFileModel`. My visual representation could be a file-upload dialog.

MAMagnitudeDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`

Subclasses `MADateDescription`, `MADurationDescription`, `MANumberDescription`, `MATimeDescription`, `MATimeStampDescription`

I am an abstract description for subclasses of `Magnitude`. The range of accepted values can be limited using the accessors `min:` and `max:`.

accessing

- `max: aMagnitudeOrNil`
Set the maximum for accepted values, or `nil` if open.
- `min: aMagnitudeOrNil`
Set the minimum for accepted values, or `nil` if open.

convenience

- `min: aMinimumObject max: aMaximumObject`
Set the minimum and maximum of accepted values, or `nil` if open.

MAMemoDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAStringDescription`

I am a description of multiline strings, possible values are instances of `String`. My visual representation could be a text-area field.

MAMultipleOptionDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAReferenceDescription`, `MAOptionDescription`

I am a description of multiple options, possible options are stored within the `options` field, possible values are instances of `Collection`. My visual representation could be a multi-select list or a group of check-boxes.

MANumberDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAMagnitudeDescription`

I am a description of numbers, possible values are instances of `Number` and all its subclasses, including `Integer` and `Float`. My visual representation could be a number input-box or even a slider-control.

MAOptionDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAReferenceDescription`

Subclasses `MAMultipleOptionDescription`, `MASingleOptionDescription`

I am an abstract description of different options the user can choose from. My instance variable `options` references the options I am representing. The options can be sorted or unsorted.

MAPasswordDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAStringDescription`

I am a description of a password string, possible values are instances of `String`. My visual representation could be a password field, where there are stars printed instead of the characters the user enters.

MAPriorityContainer

Superclasses `Object`, `MAObject`, `MADescription`, `MAContainer`

I am a container holding a collection of descriptions and I keep them sorted according to their priority.

MAReferenceDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`

Subclasses `MAMultipleOptionDescription`, `MAOptionDescription`, `MARelationDescription`, `MASingleOptionDescription`, `MATableDescription`, `MATokenDescription`, `MAToManyRelationDescription`, `MAToOneRelationDescription`

I am an abstract superclass for descriptions holding onto another description.

MARelationDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAReferenceDescription`

Subclasses `MAToManyRelationDescription`, `MAToOneRelationDescription`

I am an abstract description for descriptions representing a relation. My instance variable `classes` references a collection of possible classes that I can relate to. If required the reference description will be automatically built from this list of classes.

accessing-dynamic

- `commonClass`

Answer a common superclass of the classes of the receiver. The algo-

rithm is implemented to be as efficient as possible. The inner loop will be only executed the first few iterations.

- **reference**

The reference within a `MARelationDescription` is calculated automatically from all the classes of the receiver, if set to `nil`. By setting the reference to a `MAContainer` instance it is possible to customize the reference description.

MASingleOptionDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MARelationDescription`, `MAOptionDescription`

I am a description of a single option, possible values are stored within the `options` field, but I might also be extensible so that the user can add its own option. My visual representation could be a drop-down list or a group of option-buttons.

MAStringDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`

Subclasses `MAMemoDescription`, `MAPasswordDescription`, `MASymbolDescription`

I am a description of strings, possible values are instances of `String`. My visual representation could be a single line text-field. Use `MAMemoDescription` for multi-line strings.

MASymbolDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAStringDescription`

I am a description of symbols, possible values are instances of `Symbol`.

MATableDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MARelationDescription`

I am a description of tables, their cells and labels. I hold a reference to the description of my cells, that are all described using the same description. Possible values include instances of `MATableModel`.

MATimeDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAMagnitudeDescription`

I am a description of times, possible values are instances of `Time`. My visual representation could be a time-picker.

MATimeStampDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MAMagnitudeDescription`

I am a description of timestamps, possible values are instances of `TimeStamp`. My visual representation could be a date- and time-picker.

MATokenDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MARefereceDescription`

I am a description of tokens all described by the referenced description, possible values are instances of `SequenceableCollection`.

MAToManyRelationDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MARefereceDescription`, `MARelationDescription`

I am a description of an one-to-many relationship, possible values are instances of `Collection`.

MAToOneRelationDescription

Superclasses `Object`, `MAObject`, `MADescription`, `MAElementDescription`, `MARefereceDescription`, `MARelationDescription`

I am a description of an one-to-one relationship.

Magritte-Model-Accessor

MAAccessor

Superclasses `Object`, `MAObject`

Subclasses `MAAutoSelectorAccessor`, `MABlockAccessor`, `MAChain-Accessor`, `MAContainerAccessor`, `MADictionaryAccessor`, `MANull-Accessor`, `MASelectorAccessor`, `MAVariableAccessor`

I am the abstract superclass to all accessor strategies. Accessors are used to implement different ways of accessing (reading and writing) data from instances using a common protocol: data can be uniformly read and written using `#readFrom:` respectively `#write:to:`.

model

- `read: aModel`
Read from `aModel` using the access-strategy of the receiver.
- `write: anObject to: aModel`
Write `anObject` to `aModel` using the access-strategy of the receiver.

testing

- `canRead: aModel`
Test if `aModel` can be read.
- `canWrite: aModel`
Test if `aModel` can be written.

MAAutoSelectorAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`, `MASelector-Accessor`

I am very similar to my super-class `MASelectorAccessor`, however I do create instance variables and accessor methods automatically if necessary. I am especially useful for prototyping. I never change existing accessor methods.

MABlockAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`

I am an access strategy defined by two block-closures. The read-block expects the model as its first argument and is used to retrieve a value. The write-block expects the model as its first and the value as its second argument and is used to write a value to the model.

MAChainAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`

I am an access strategy used to chain two access strategies. To read and write a value the `accessor` is performed on the given model and the result is passed into the `next` accessor.

MAContainerAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`

I am a read-only access strategy and I answer the model itself when being read.

MADictionaryAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`

I am an access strategy to be used on dictionaries. I use my `key` to read from and write to indexed collections.

MANullAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`

I am a null access strategy and I should be neither read nor written. I am still comparable to other strategies by holding onto a unique-identifier.

MASelectorAccessor

Superclasses `Object`, `MAObject`, `MAAccessor`

Subclasses `MAAutoSelectorAccessor`

I am the most common access strategy defined by a read- and a write-selector. I am mostly used together with standard getters and setters as usually defined by the accessing protocol. If there is only a read-selector specified, the write selector will be deduced automatically by adding a colon to the read-selector.

MAVariableAccessor**Superclasses** `Object`, `MAObject`, `MAAccessor`

I am an access strategy that directly reads from and writes to instance variables. I strongly violate encapsulation and most of the time I should be replaced by an instance of `MASelectorAccessor`.

Magritte-Model-Condition**MAAllCondition****Superclasses** `Object`, `MAObject`, `MACondition`, `MAComposed-Condition`

I am a condition that is satisfied if *all* of my child-conditions are satisfied.

MAAnyCondition**Superclasses** `Object`, `MAObject`, `MACondition`, `MAComposed-Condition`

I am a condition that is satisfied if *any* of my child-conditions are satisfied.

MACondition**Superclasses** `Object`, `MAObject`**Subclasses** `MAAllCondition`, `MAAnyCondition`, `MAFalseCondition`, `MANoneCondition`, `MASelectorCondition`, `MATrueCondition`

I am an abstract condition. To check if the condition is satisfied by a given model-object, send the message `#value:` to myself.

evaluation

- `numArgs`
Answer the number of arguments that must be used to evaluate this condition.
- `value: anObject`
Evaluate the receiver with the argument `anObject`. Answer a `Boolean` telling if the condition is met for `anObject`.

operators-binary

- `& aCondition`
The resulting condition will require the receiver *and* `aCondition` to be satisfied.
- `| aCondition`
The resulting condition will require the receiver *or* `aCondition` to be satisfied.

operators-unary

- `not`
Negates the receiving condition.

MAFalseCondition

Superclasses `Object`, `MAObject`, `MACondition`, `MAConstantCondition`

I am a condition that is satisfied for *no* input. I always answer `false`.

MANoneCondition

Superclasses `Object`, `MAObject`, `MACondition`, `MAComposedCondition`

I am a condition that is satisfied if *none* of my child-conditions are satisfied.

MASelectorCondition

Superclasses Object, MAObject, MACondition

I am a condition that performs a specific selector on the model to test if a condition is satisfied or not.

MATrueCondition

Superclasses Object, MAObject, MACondition, MAConstant-Condition

I am a condition that is satisfied for *all* input. I always answer `true`.

Magritte-Model-Memento

MACachedMemento

Superclasses Object, MAObject, MAMemento

Subclasses MACheckedMemento

I cache values being read and written without touching the model. When committing changes, the modifications will be propagated to the model all at once.

testing

- `hasChanged`
Answer `true`, if the cached data is different to the data in the model.

MACheckedMemento

Superclasses Object, MAObject, MAMemento, MACachedMemento

I cache values as my superclass and also remember the original values of the model at the time the cache is built. With this information I am able to detect edit conflicts and can prevent accidental loss of data by merging the changes.

testing

- `hasConflict`
Answer `true`, if there is an edit conflict.

MAMemento

Superclasses `Object`, `MAObject`

Subclasses `MACachedMemento`, `MACheckedMemento`, `MAStraitMemento`

I am an abstract memento. I reference a model I am working on and the description currently used to describe this model.

actions

- `commit`
Commit the receiver into the model.
- `reset`
Reset the memento from the model.
- `validate`
Check if the data in the receiver would be valid if committed. In case of problems an exception is raised.

MAStraitMemento

Superclasses `Object`, `MAObject`, `MAMemento`

I am a memento that forwards read- and write-access directly to the model. I can mostly be replaced with the model itself.

Magritte-Model-Exception

MAConditionError

Superclasses `Object`, `Exception`, `Error`, `MAError`, `MAValidationError`

I am an error that is raised whenever a user-defined condition is failing.

MAConflictError

Superclasses Object, Exception, Error, MAError, MAValidationError

I am an error that is raised whenever there is an edit conflict.

MAError

Superclasses Object, Exception, Error

Subclasses MAConditionError, MAConflictError, MAKindError, MAMultipleErrors, MARangeError, MAREadError, MAREquiredError, MAValidationError, MAWriteError

I represent a generic Magritte error.

MAKindError

Superclasses Object, Exception, Error, MAError, MAValidationError

I am an error that is raised whenever a description is applied to the wrong type of data.

MAMultipleErrors

Superclasses Object, Exception, Error, MAError, MAValidationError

I am an error that is raised whenever there are multiple validation rules failing.

MARangeError

Superclasses Object, Exception, Error, MAError, MAValidationError

I am an error that is raised whenever a described value is out of bounds.

MAReadError

Superclasses `Object`, `Exception`, `Error`, [MAError](#)

I am an error that gets raised when there is problem reading serialized data.

MARequiredError

Superclasses `Object`, `Exception`, `Error`, [MAError](#), [MAValidationError](#)

I am an error that is raised whenever a required value is not supplied.

MAValidationError

Superclasses `Object`, `Exception`, `Error`, [MAError](#)

Subclasses [MAConditionError](#), [MAConflictError](#), [MAKindError](#), [MAMultipleErrors](#), [MARangeError](#), [MARequiredError](#)

I am a generic validation error. I reference the description that caused the validation error.

MAWriteError

Superclasses `Object`, `Exception`, `Error`, [MAError](#)

I am an error that gets raised when there is problem writing serialized data.

Magritte-Model-Visitor

MAVisitor

Superclasses `Object`

I am a visitor responsible to visit Magritte descriptions. I am an abstract class providing a default implementation for concrete visitors. The protocol I am implementing reflects the hierarchy of [MADescription](#) with its subclasses so that visiting a specific class automatically calls less specific

implementations in case the specific implementation has been left out. The code was automatically created using code on my class-side.

visiting

- `visitAll: aCollection`
Visit all elements of `aCollection` with the receiving visitor.
- `visit: anObject`
Visit `anObject` with the receiving visitor.

Magritte-Model-Utility

MADynamicObject

Superclasses `MAProxyObject`

A dynamic object can be used for almost any property within Magritte that is not static but calculated dynamically. This is a shortcut to avoid having to build context sensitive descriptions manually over and over again, however there are a few drawbacks:

- Some messages sent to this proxy, for example `#class` and `#value`, might not get resolved properly.
- Raising an unhandled exception will not always open a debugger on your proxy, because tools are unable to properly work with the invalid object and might even crash your image.

MANamedBuilder

Superclasses `Object`, `MADescriptionBuilder`

I dynamically build container descriptions from class-side methods using a simple naming convention for the selector names:

1. The method `#defaultContainer` is called to retrieve the container instance.
2. All the unary methods starting with the selector `#description` are called and should return a valid description to be added to the container.

3. All the keyword messages with one argument having a prefix of a method selected in step 2 will be called with the original description to further refine its definition.

MAPragmaBuilder

Superclasses `Object`, `MADescriptionBuilder`

I dynamically build container descriptions defined statically in classes using all the methods being tagged with the pragmas `description` or `description:.` I only work with Smalltalk implementations that have a decent implementation of Pragmas, such as Squeak 3.9.

MAProxyObject

Subclasses `MADynamicObject`

I represent an abstract proxy object, to be refined by my subclasses.

copying

- `copy`
It doesn't make sense to copy proxies in most cases, the real-subject needs to be looked up and will probably return a new instance on every call anyway.

printing

- `printOn: aStream`
Print the receiver on `aStream` but within square-brackets to show that it is a proxied instance.

testing

- `isNil`
This method is required to properly return `true` if the `realSubject` is `nil`.

Pier-Model-Core

PRObject

Superclasses Object

Subclasses PRAncor, PRChildren, PRCommand, PRContext, PRDecorated, PRDecoration, PRDocument, PRDocumentGroup, PRDocumentItem, PRExternalLink, PRFile, PRHeader, PRHider, PRHorizontalRule, PRInternalLink, PRIsbnLink, PRKernel, PRLink, PRList, PRListItem, PRMailLink, PROrderedList, PRPage, PRParagraph, PRPreformatted, PRRfcLink, PRStructure, PRTable, PRTableCell, PRTableRow, PRText, PRUnorderedList

I am the root of objects within Pier. I hold a dictionary of properties, so that users can easily annotate me with new values. I am visitable.

accessing-properties

- **properties**
Answer the property dictionary of the receiver.
- **propertyAt: aKey**
Answer the value of the property **aKey**, raises an error if the property doesn't exist.
- **propertyAt: aKey ifAbsentPut: aBlock**
Answer the value of the property **aKey**, or if the property doesn't exist adds and answers the result of evaluating **aBlock**.
- **propertyAt: aKey ifAbsent: aBlock**
Answer the value of the property **aKey**, or the result of **aBlock** if the property doesn't exist.
- **propertyAt: aKey put: aValue**
Adds or replaces the property **aKey** with **aValue**.

testing

- **hasProperty: aKey**
Test if the property **aKey** is defined within the receiver.

visiting

- `accept: aVisitor`
Dispatch to `aVisitor` depending on the receiver.

Pier-Model-Kernel

PRContext

Superclasses `Object`, `PRObject`

I am the context in which a user is browsing the system. I hold all the information any part of Pier might be interested in: the currently used kernel, the structure that is currently displayed, the command that is being executed and the user currently logged in.

I am an immutable object. Users should never try to modify me. Instead use the modification methods that return a copy of myself.

accessing

- `command`
Answer the active command of this context.
- `enumerator`
Answer a default structure enumerator for the current context.
- `kernel`
Answer the underlying kernel of this context.
- `structure`
Answer the currently browsed structure of this context.

accessing-convenience

- `commands`
Answer a list of possible commands, dispatching through the command class.
- `root`
Answer the current root node of the structure-tree.
- `timestamp`
Answer the timestamp when this context was used to execute its command, `nil` if never executed.

enumerating

- **enumerator: aStructure**
Answer an enumerator on **aStructure**.

navigation

- **command: aCommand**
Create a copy of the current context with the current command replaced by **aCommand**.
- **structure: aStructure**
Create a copy of the current context with the current structure replaced by **aStructure** and the current command replaced by the default view.
- **structure: aStructure command: aCommand**
Create a copy of the current context with the current structure replaced by **aStructure** and the current command by **aCommand**.

testing

- **isValid**
Answer **true** if the receiver is a valid context.
- **isValidCommand: aCommandClass**
Answer **true** if the receiver is a valid context with the current command replaced by **aCommandClass**.

PRCurrentContext

Superclasses Object, Exception, Notification

I am a dynamic variable. I answer the current context when being raised.

PRKernel

Superclasses Object, [PRObject](#)

I am the kernel of Pier. Several instances of myself might exist at the same time, but they all exist independently and don't share any data. I know the root structure and the persistency strategy of the whole data-model.

Moreover I prevent any concurrent modifications to the model by providing a global mutex.

accessing

- **name**
Answer the name of the kernel.
- **persistence**
Answer the persistence strategy of the receiver.
- **root**
Answer the root structure of the kernel.

accessing-readonly

- **mutex**
Return a mutex (an object that understands `#critical:`) to ensure that only one process is modifying the model at once. This is needed to make certain that the model remains in a consistent state. All write access must go through this mutex.

Pier-Model-Structure

PRChildren

Superclasses `Object`, `PRObject`, `PRDecoration`

I hold the children of the decorated object.

accessing

- **size**
Answer the number of children of the receiver.

accessing-children

- **at: aString**
Answer the child structure with the name `aString`, raise an error if the child can't be found.

- `at: aString ifAbsent: aBlock`
Answer the child structure with the name `aString`, evaluate `aBlock` if the child can't be found.

actions

- `add: aStructure`
Add `aStructure` as a child to the receiver.
- `remove: aStructure`
Remove `aStructure` from the receiver.

PRDecorated

Superclasses `Object`, `PRObject`

Subclasses `PRFile`, `PRPage`, `PRStructure`

I am an abstract decorated object. My decorations are subclasses of `PRDecoration`. I provide all the tools to add, remove, query and visit my decorations.

accessing

- `decorations`
Answer the sorted decorations of the receiver.

adding

- `addDecoration: aDecoration`
Add `aDecoration` to the receiver. This message ensures that `aDecoration` is only added once and that the decorations remain properly sorted.
- `addDecoration: aDecoration ifPresent: aBlock`
Add `aDecoration` to the receiver. This message ensures that the decorations remain properly sorted and that there are no duplicates. In case `aDecoration` is already within the receiver, the existing decoration is passed into `aBlock`.

enumerating

- **decorationsDo: aBlock**
Evaluate **aBlock** in the right order with each of the receiver's decorations as the argument.
- **decorationsDo: aBlock ownerDo: anOwnerBlock**
Evaluate **aBlock** and **anOwnerBlock** in the right order with each of the receiver's decorations and the receiver as the argument.

querying

- **decorationOfClass: aClass**
Answer the first decoration of **aClass**, raise an error if none could be found.
- **decorationOfClass: aClass ifAbsent: aBlock**
Answer the first decoration of **aClass**, evaluate **aBlock** if none could be found.

removing

- **removeDecoration: aDecoration**
Remove **aDecoration** from the receiver, an error is raised if **aDecoration** is not part of the receiver.
- **removeDecoration: aDecoration ifAbsent: aBlock**
Remove **aDecoration** from the receiver, **aBlock** is evaluated if **aDecoration** is not part of the receiver.

PRDecoration

Superclasses `Object`, `PRObject`

Subclasses `PRChildren`, `PRHider`

I am an abstract decoration to add new behaviour and data to structures. Every decoration knows its owner, a subclass of `PRDecorated`. Decorations are considered to be equal if they are of the same species, but subclasses might want to refine this behaviour to be able to add multiple instances of the same class.

Within the owner decorations are ordered according to their priority. Decorations with a negative priority are visited before the owner, decorations with a positive one after the owner.

accessing

- **decorated**
Answer the owner of the receiver, the decorated object.
- **priority**
The default priority returns a number that defines in witch order the visitors will traverse through decorated objects. Negative numbers are visited before the decorated objects, positive numbers afterwards.

testing

- **isAllowedCommand: aCommandClass in: aContext**
Answer **true** if the receiver allows one to execute **aCommandClass** in **aContext**. The default decoration is fine with all the commands, subclasses might restrict to a selected set of commands within a given context. This method might be overridden by decorations that want to control the security.

PRFile

Superclasses `Object`, `PRObject`, `PRDecorated`, `PRStructure`

I represent a data container for images, videos, sound, pdf or zip files. I reference an instance of `MAFileModel`. The mime-type is used to determine how the given file is be displayed. As an example images and videos are attempted to be inlined into the resulting output, whereas zip-files are referenced as a link to allow downloading.

PRHider

Superclasses `Object`, `PRObject`, `PRDecoration`

I hide the decorated object.

PRPage

Superclasses `Object`, `PRObject`, `PRDecorated`, `PRStructure`

I am the most important class of the structure hierarchy. I reference a composite of documents modeling the contents of the page that the user entered using the Wiki syntax. When initializing the instance a default document will be created to make the user aware of the newly created page.

accessing

- **document**
Answer the document of the receiver.

accessing-configuration

- **defaultDocument**
Answer the default document of the receiver.
- **parserClass**
Answer the default document parser for the receiver.
- **rendererClass**
Answer the default document writer for the receiver.

PRStructure

Superclasses `Object`, `PRObject`, `PRDecorated`

Subclasses `PRFile`, `PRPage`

I am an abstract structure, representing the model of a single page. I've got a name, that must be unique within the scope of my parent, and a title. A structure is identified with a path of structure names.

accessing

- **name**
Answer the name of the receiver. The name should be simple and only contain letters and numbers, since it is used as an identifier within restrictive protocols.
- **parent**
Answer the parent structure of the receiver.

- **title**
Answer the title of the receiver, essentially the name but starting uppercase.

accessing-children

- **addChild: aStructure**
Add **aStructure** as child to the receiver.
- **enumerator**
Answer an enumerator on the children of the receiver.

accessing-commands

- **editCommandClass**
Answer an instance of the default edit command of the receiver.
- **viewCommandClass**
Answer the default view command of the receiver. Most likely you never need to change the default implementation.

accessing-dynamic

- **icon**
Return the raw data of an icon representing the type of the receiver.
- **kernel**
Answer the kernel of the receiver.
- **level**
Answer the nesting level of the receiver.
- **parents**
Answer an ordered collection of all the parents of the receiver up and including the receiver itself.
- **root**
Answer the root structure of the receiver.

accessing-persistency

- **creationTimestamp**
Answer the creation-timestamp or **nil**.

- `modificationTimestamp`
Answer the modification-timestamp or `nil`.

actions

- `remove`
Remove the receiver from the parent structure.

decorations

- `childrenDecoration`
Answer a decoration with the children of the receiver. If no children exist, an empty children decoration is added to the receiver.

testing

- `canBeChildOf: aStructure`
Answer `true` if the receiver can be a child of `aStructure`.
- `canBeParentOf: aStructure`
Answer `true` if the receiver can be a parent of `aStructure`.
- `hasChildren`
Answer `true` if the receiver has got children, they might not be visible however.
- `hasParent`
Answer `true` if the receiver has got a parent. This is the negation of `#isRoot`.
- `isAllowedCommand: aCommandClass in: aContext`
Answer `true` if the receiver accepts `aCommandClass` as allowed in `aContext` on the receiver. If a command class is allowed is a security question and should therefore only depend on the permissions in the current context. The default implementation delegates the decision to the decorations of the receiver. Therefor this method should never be overridden by subclasses.
- `isAncestorOf: aStructure`
Answer `true` if the receiver is an ancestor of `aStructure`.
- `isApplyableCommand: aCommandClass in: aContext`
Answer `true` if the receiver accepts `aCommandClass` as applyable in `aContext` on the receiver. If a command class is applyable or not is a question of saneness and compatibility, not of security. This method

should be overridden by subclasses who want to forbid some commands.

- **isRoot**
Answer **true** if the receiver is the root of the Pier model. This is the negation of **#hasParent**.
- **isValidCommand: aCommandClass in: aContext**
Answer **true** if the receiver accepts to execute **aCommandClass** in **aContext**. This message is the combination of **#isApplicableCommand:in:** and **#isAllowedCommand:in:**. It should not be overridden.

Pier-Model-Document

PRAnchor

Superclasses Object, [PRObject](#), [PRDocumentItem](#)

I am an anchor within a document. I am used as a reference point within a large document.

PRDocument

Superclasses Object, [PRObject](#), [PRDocumentItem](#), [PRDocumentGroup](#)

I am the root of a document composite.

PRDocumentGroup

Superclasses Object, [PRObject](#), [PRDocumentItem](#)

Subclasses [PRDocument](#), [PRExternalLink](#), [PRHeader](#), [PRInternalLink](#), [PRIsbnLink](#), [PRLink](#), [PRList](#), [PRListItem](#), [PRMailLink](#), [PROrderedList](#), [PRParagraph](#), [PRPreformatted](#), [PRRfcLink](#), [PRTTable](#), [PRTTableCell](#), [PRTTableRow](#), [PRUnorderedList](#)

I am an abstract group of document items.

accessing

- **children**
Answer the children of the receiver.

PRDocumentItem

Superclasses `Object`, `PRObject`

Subclasses `PRAnchor`, `PRDocument`, `PRDocumentGroup`, `PRExternalLink`, `PRHeader`, `PRHorizontalRule`, `PRInternalLink`, `PRIsbnLink`, `PRLink`, `PRList`, `PRListItem`, `PRMailLink`, `PROrderedList`, `PRParagraph`, `PRPreformatted`, `PRRfcLink`, `PRTable`, `PRTableCell`, `PRTableRow`, `PRText`, `PRUnorderedList`

I am an abstract superclass for the document hierarchy. My subclasses include all the basic elements to represent a document.

accessing

- **owner**
Answer the object owning the receiver. The default implementation doesn't know about its owner and therefore always returns `nil`.

accessing-dynamic

- **text**
Answer a string representation of the receiver.

PRDocumentParser

Superclasses `Object`, `SmaCCParser`

I am a parser that builds a composite of document-items from a Wiki string. I don't raise errors for invalid input, but instead try to build a parse tree that is as close to the source as possible. I am automatically generated from the Smalltalk Compiler Compiler (SmaCC), do not edit my code manually.

Example

```
(PRDocumentParser parse: '!Foo bar')
  explore
```

PRDocumentWriter

Superclasses `Object`, `PRVisitor`

I am a visitor being able to transform a composite of document-items back into the original Wiki string.

Example

```
| document |
document := PRDocumentParser parse: '+Hello+ *World*'.
string := PRDocumentWriter write: document.
string inspect.
```

visiting-document

- `visitText: anObject`
Write out `anObject`'s text to the receivers output-stream and escape `$*` everywhere within. Also take care of the character escaping with `$=`, `$|`, `$!`, `$#` and `$-` at the beginning of a line.

PRExternalLink

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`, `PRLink`

Subclasses `PRIsbnLink`, `PRMailLink`, `PRRfcLink`

I am an abstract external link with an URL (Uniform Resource Locator) as reference.

accessing

- `url`
Answer the URL the receiver is pointing to.

PRHeader

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`

I represent a header within a document. My level is a natural number.

PRHorizontalRule

Superclasses `Object`, `PRObject`, `PRDocumentItem`

I am a horizontal rule.

PRInternalLink

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`, `PRLink`

I am an internal link pointing to a structure within the current kernel. I reference my owning structure to be able to lookup the referenced structure. The referenced structure is cached in the instance variable `target`.

accessing

- **anchor**
Answer an anchor string the receiver is pointing to.
- **target**
Answer the referenced structure.

actions

- **refresh**
This message will be sent by the structure whenever it is parsed and the references have to be set up. It simply starts a look-up in the owner using the reference-string. In case the reference is invalid the target will be set to `nil` and the receiver is in a broken-state.
- **update**
This method will be sent to all the internal-links whenever the owner is renamed or moved to a different location in the structure tree. It automatically adjusts the receivers state, so that the reference still points to the right location and doesn't get broken.

PRIsbnLink

Superclasses Object, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`, `PRLink`, `PRExternalLink`

I am an external link pointing to an ISBN (International Standard Book Number).

PRLink

Superclasses Object, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`

Subclasses `PRExternalLink`, `PRInternalLink`, `PRIsbnLink`, `PRMailLink`, `PRRfcLink`

I am an abstract link built from an alias and a reference. The alias is the string representation that will be displayed to the user, whereas the reference is a string identifying the target. If there is no alias, the reference itself is displayed. Links can try to embed the referenced target into the containing document.

Examples

```
*Reference*
+Alias>Embedded Reference*
```

accessing

- **embedded**
Answer `true` if the reference should be embedded.
- **reference**
Answer the reference of the receiver.

accessing-dynamic

- **alias**
Answer the alias of the receiver or an empty string if none.

testing

- `isBroken`
Answer `true` if the receiver is broken.

PRList

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`

Subclasses `PROrderedList`, `PRUnorderedList`

I am an abstract list. My children are instances of `PRListItem`.

PRListItem

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`

I am a item within a `PRList`.

PRMailLink

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`, `PRLink`, `PRExternalLink`

I am an external link pointing to a mail address. I encode my URL to prevent spam bots collecting the address.

PROrderedList

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`, `PRList`

I am an ordered list. I am typically used for numbered items.

PRParagraph

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`

I am a paragraph of text.

PRPreformatted

Superclasses Object, [PObject](#), [PRDocumentItem](#), [PRDocumentGroup](#)

I am preformatted text or source code. My children are instances of [PRText](#).

PRRfcLink

Superclasses Object, [PObject](#), [PRDocumentItem](#), [PRDocumentGroup](#), [PRLink](#), [PRExternalLink](#)

I am an external link pointing to a RFC (Request for Comments) document.

PRTable

Superclasses Object, [PObject](#), [PRDocumentItem](#), [PRDocumentGroup](#)

I am a table. My children are instances of [PRTableRow](#).

PRTableCell

Superclasses Object, [PObject](#), [PRDocumentItem](#), [PRDocumentGroup](#)

I am a cell of a table.

PRTableRow

Superclasses Object, [PObject](#), [PRDocumentItem](#), [PRDocumentGroup](#)

I am a row of a table. My children are instances of [PRTableCell](#).

PRText

Superclasses Object, [PObject](#), [PRDocumentItem](#)

I am a plain text. I am the most important leaf node of the document composite.

accessing

- `text`
Answer the string the receiver is representing.

PRUnorderedList

Superclasses `Object`, `PRObject`, `PRDocumentItem`, `PRDocumentGroup`, `PRList`

I am an unordered list. I am typically used for unnumbered lists

Pier-Model-Command

PRCommand

Superclasses `Object`, `PRObject`

I am an abstract superclass of the command pattern in Pier. All modifications to the model have to be done through subclasses of myself, else they do not get properly logged with the persistency mechanism. I hold the context in which the receiver is executed in the instance variable `context`. To modify the resulting context, create a copy of the current context and store it in the instance variable `answer`.

The following event-handlers are called when executing an action in the given order. Do override these messages to customize the command, never override the other internal methods:

- Override the message `#doValidate` to check the valid setup of the command and to raise exceptions in case any precondition isn't met. Speak here or forever have your peace! Don't change the model in there.
- Override the message `#doExecute` to execute the actual command. Do not raise exceptions in there, catch all the problems in `#doValidate`.
- Override the message `#doPersistency` to save the command that has been just executed with the current persistency strategy. Most commands don't need to override this message and just stick with the default behavior.
- Override the message `#doAnswer` to create the answer context. Most commands don't need to override this message and just stick with the default behavior.

Do not play with funny jumpy things, such as resumable exceptions or continuations, inside the code of the command hierarchy or you will very likely run into severe problems.

accessing

- **answer**
Return a new context that should be activated after executing this action.
- **answer: aContext**
Set the resulting context of this command.
- **context**
Return the current context of the receiver.
- **context: aContext**
Set the current context of the receiver.

accessing-delegated

- **kernel**
Answer the kernel the receiver is working on.

actions

- **execute**
Execute the command of the receiver. To implement your action in the code of one of my subclasses. Never override this message, but instead have a look at the different template methods (**#doValidate**, **#doExecute**, **#doPersistency**, **#doAnswer**) available in the events protocol.

events

- **doAnswer**
Override this message to create the answer context. Most commands don't need to override this message and just stick with the default behavior.
- **doExecute**
Override this message to execute the actual command. Do not raise exceptions in there, catch all the problems in **#doValidate**.

- **doPersistency**
Override this message to save the command-context that has been just executed with the current persistency strategy. Most commands don't need to override this message and just stick with the default behavior.
- **doValidate**
Override this message to check the valid setup of the command and to raise exceptions in case any precondition isn't met. Speak here or forever have your peace! Don't change the model in there.

testing

- **isLogged**
Most commands are logged. This means they do modify the model and are therefore preserved in the history. However there are some that just display something or change the state of the current context and neither change the model nor need to be logged.
- **isQuick**
Quick commands do not have a configuration interface (probably not even Magritte descriptions) and therefore should not be presented to the user but simply executed.
- **isView**
Most commands are not a view. This means they override `#doExecute` to do something meaningful on the context or the model.

validating

- **validateChild: aChildStructure in: aStructure**
Make sure that `aChildStructure` can be added as child or replace an existing child in a `aStructure`.
- **validateName: aString of: aChildStructure in: aStructure**
Make sure that `aChildStructure` with the title `aString` can be added as child or replace an existing child in a `aStructure`.
- **validateNestingOf: aChildStructure in: aStructure**
Make sure that `aChildStructure` can be added as child to `aStructure` and that `aStructure` can have `aChildStructure` as a child.

Pier-Model-Visitor

PRFullTextSearch

Superclasses Object, [PRVisitor](#), PRPluggableVisitor

I am a pluggable visitor to quickly look for matching text in a subtree of structures.

Example

The following example opens an inspector on all structures with the text foo:

```
(PRFullTextSearch
  from: aStructure
  find: 'foo'
  caseSensitive: false)
  inspect
```

PRIncomingReferences

Superclasses Object, [PRVisitor](#), PRPluggableVisitor

I am a pluggable visitor to detect incoming references.

Example

The following example opens inspectors on all instances of [PRInternalLink](#) that point aStructure:

```
PRIncomingReferences
  from: aRootStructure
  to: aStructure
  do: [ :each | each inspect ]
```

PROutgoingReferences

Superclasses Object, [PRVisitor](#), PRPluggableVisitor

I am a pluggable visitor visiting and eventually following outgoing references. To do so I visit all the links of the page and evaluate my pluggable block for each of them, if the block answers `true` I follow the link and continue visiting the references of the target structure. I take care not to run into infinite recursion, so no structure will be visited more than once.

Example

The following example opens an inspector on all the broken references that are seen when displaying `aStructure`:

```
PROutgoingReferences
  start: aStructure
  do: [ :each |
    each isBroken
      ifTrue: [ each inspect ].
    each isEmbedded ].
```

PRPathLookup

Superclasses `Object`, `PRVisitor`, `PRPath`

I am a visitor used to look up a given path. I am able to lookup absolute and relative paths, following the syntax of unix operating systems.

Example

```
(PRPathLookup
  start: aStructure
  path: '/Information/Copyright/..')
  inspect
```

visiting-decorations

- `visitChildren: anObject`

This method does the lookup of the next structure by checking for a child with that name, since this value is hashed it can be done efficiently. If the lookup by name fails, it tries to match the title by iterating through the children. If this fails as well, the message `#childNotFound:` is sent.

PRPathReference

Superclasses `Object`, `PRVisitor`, `PRPath`

I am a visitor used to print a short path from a structure to another one.

Example

```
(PRPathReference
  from: aFirstStructure
  to: aSecondStructure)
inspect
```

PRVisitor

Superclasses Object

Subclasses PRDocumentWriter, PRFullTextSearch, PRIncomingReferences, PROutgoingReferences, PRPathLookup, PRPathReference

I am an abstract visitor. I provide a default implementation of all visit messages that does not descend automatically into children of the visited graph. Subclasses should override all my messages in appropriate ways to visit the nodes they need.

Index

`<=`, 61
`=`, 55
`&`, 72

`accept:`, 80
`Accessor`, 69
`accessor`, 60
`AdaptiveModel`, 55
`add:`, 83
`addChild:`, 87
`addCondition:labelled:`, 62
`addDecoration:`, 83
`addDecoration:ifPresent:`, 83
`alias`, 93
`AllCondition`, 71
`Anchor`, 89
`anchor`, 92
`answer`, 97
`answer:`, 97
`AnyCondition`, 71
`asContainer`, 61
`at:`, 82
`at:at:`, 57
`at:at:put:`, 57
`at:ifAbsent:`, 83
`AutoSelectorAccessor`, 69

`BlockAccessor`, 70
`BooleanDescription`, 57

`CachedMemento`, 73
`canBeChildOf:`, 88
`canBeParentOf:`, 88
`canRead:`, 69
`canWrite:`, 69
`ChainAccessor`, 70
`CheckedMemento`, 73

`Children`, 82
`children`, 90
`childrenDecoration`, 88
`ClassDescription`, 58
`ColorDescription`, 58
`columnCount`, 57
`Command`, 96
`command`, 80
`command:`, 81
`commands`, 80
`comment`, 60
`commit`, 74
`commonClass`, 66
`Compatibility`, 53
`Condition`, 71
`ConditionError`, 74
`conditions`, 60
`ConflictError`, 75
`Container`, 58
`ContainerAccessor`, 70
`Context`, 80
`context`, 97
`context:`, 97
`copy`, 78
`creationTimestamp`, 87
`CurrentContext`, 81

`DateDescription`, 58
`Decorated`, 83
`decorated`, 85
`Decoration`, 84
`decorationOfClass:`, 84
`decorationOfClass:ifAbsent:`,
84
`decorations`, 83
`decorationsDo:`, 84

- decorationsDo:ownerDo:, 84
- defaultDocument, 86
- Description, 59
- description, 55
- DictionaryAccessor, 70
- Distribution, 53
- doAnswer, 97
- Document, 89
- document, 86
- DocumentGroup, 89
- DocumentItem, 90
- DocumentParser, 90
- DocumentWriter, 91
- doExecute, 97
- doPersistency, 98
- doValidate, 98
- dump, 53
- DurationDescription, 63
- DynamicObject, 77
- editCommandClass, 87
- ElementDescription, 63
- embedded, 93
- enumerator, 80, 87
- enumerator:, 81
- Error, 75
- execute, 97
- extension, 56
- ExternalLink, 91
- FalseCondition, 72
- File, 85
- FileDescription, 64
- FileModel, 56
- fromString:, 62
- fromString:reader:, 62
- fromStringCollection:, 61
- fromStringCollection:reader:, 62
- FullTextSearch, 99
- hasChanged, 73
- hasChildren, 62, 88
- hasComment, 62
- hasConflict, 74
- hash, 54
- hasLabel, 62
- hasParent, 88
- hasProperty:, 55, 79
- Header, 92
- Hider, 85
- HorizontalRule, 92
- icon, 87
- IncomingReferences, 99
- InternalLink, 92
- isAllowedCommand:in:, 85, 88
- isAncestorOf:, 88
- isApplication, 56
- isApplicableCommand:in:, 88
- isAudio, 56
- IsbnLink, 93
- isBroken, 94
- isContainer, 62
- isDescription, 62
- isImage, 56
- isLogged, 98
- isNil, 78
- isQuick, 98
- isRoot, 89
- isSatisfiedBy:, 63
- isText, 56
- isValid, 81
- isValidCommand:, 81
- isValidCommand:in:, 89
- isVideo, 57
- isView, 98
- Kernel, 81
- kernel, 80, 87, 97
- kind, 60
- KindError, 75
- label, 60
- level, 87
- Link, 93
- List, 94
- ListItem, 94
- MagnitudeDescription, 64

- MailLink, 94
- maintype, 56
- max:, 64
- Memento, 74
- MemoDescription, 65
- min:, 64
- min:max:, 64
- modificationTimestamp, 88
- MultipleErrors, 75
- MultipleOptionDescription, 65
- mutex, 82
- name, 60, 82, 86
- NamedBuilder, 77
- NoneCondition, 72
- not, 72
- NullAccessor, 70
- numArgs, 72
- NumberDescription, 65
- Object, 53, 79
- OptionDescription, 65
- OrderedList, 94
- OutgoingReferences, 99
- owner, 90
- Page, 85
- Paragraph, 94
- parent, 86
- parents, 87
- parserClass, 86
- PasswordDescription, 65
- PathLookup, 100
- PathReference, 100
- persistency, 82
- persistent, 60
- postCopy, 55
- PragmaBuilder, 78
- Preformatted, 95
- printOn:, 78
- priority, 60, 85
- PriorityContainer, 66
- properties, 54, 79
- propertyAt:, 54, 79
- propertyAt:ifAbsent:, 54, 79
- propertyAt:ifAbsentPut:, 54, 79
- propertyAt:put:, 54, 79
- ProxyObject, 78
- publish, 53
- RangeError, 75
- read:, 69
- ReadError, 76
- readonly, 60
- readUsing:, 55
- reference, 67, 93
- ReferenceDescription, 66
- refresh, 92
- RelationDescription, 66
- remove, 88
- remove:, 83
- removeDecoration:, 84
- removeDecoration:ifAbsent:, 84
- rendererClass, 86
- required, 61
- RequiredError, 76
- reset, 74
- reshapeRows:columns:, 57
- RfcLink, 95
- root, 80, 82, 87
- rowCount, 57
- SelectorAccessor, 70
- SelectorCondition, 73
- SingleOptionDescription, 67
- size, 56, 82
- StraitMemento, 74
- StringDescription, 67
- stringReader, 61
- stringWriter, 61
- Structure, 86
- structure, 80
- structure:, 81
- structure:command:, 81
- subtype, 56
- SymbolDescription, 67

Table, 95
TableCell, 95
TableDescription, 67
TableModel, 57
TableRow, 95
target, 92
Text, 95
text, 90, 96
TimeDescription, 68
timestamp, 80
TimeStampDescription, 68
title, 87
TokenDescription, 68
ToManyRelationDescription,
68
ToOneRelationDescription, 68
toString:, 62
toString:writer:, 62
toStringCollection:, 62
toStringCollection:writer:, 62
TrueCondition, 73

undefined, 61
UnorderedList, 96
update, 92
url, 91

validate, 74
validate:, 63
validateChild:in:, 98
validateConditions:, 63
validateKind:, 63
validateName:of:in:, 98
validateNestingOf:in:, 98
validateRequired:, 63
validateSpecific:, 63
ValidationError, 76
value:, 72
values, 55
VariableAccessor, 71
viewCommandClass, 87
visible, 61
visit:, 77
visitAll:, 77
visitChildren:, 100
Visitor, 76, 101
visitText:, 91

write:to:, 69
write:using:, 56
WriteError, 76

Bibliography

- [Alpe98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.
- [Atki87] Bill Atkinson. “HyperCard”. 1987. Hypercard.
- [Brana] John Brant and Don Roberts. “SmaCC, a Smalltalk Compiler-Compiler”. <http://www.refactory.com/Software/SmaCC/>.
- [Branb] John Brant and Don Roberts. “#Smalltalk (Sharp Smalltalk)”. <http://www.refactory.com/Software/SharpSmalltalk/>.
- [Budi03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.
- [Ceri00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. “Web modeling language (WebML): a modeling language for designing Web sites”. In: *Ninth International World Wide Web Conference*, 2000.
- [Clar04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. “Applied Metamodelling: A foundation for Language Driven Development”. 2004.
- [Duca00] Stéphane Ducasse and Florence Ducasse. “De l’enseignement de concepts informatiques”. *Journal de l’association EPI Enseignement Public et Informatiques*, Vol. 4, No. 97, Sep. 2000.
- [Duca04] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. “Seaside — a Multiple Control Flow Web Application Framework”. In: *Proceedings of 12th International Smalltalk Conference (ISC’04)*, pp. 231–257, Sep. 2004.
- [Duca05] Stéphane Ducasse, Lukas Renggli, and Roel Wuyts. “SmallWiki — A Meta-Described Collaborative Content Management System”. In: *Proceedings ACM International Symposium on Wikis*

- (*WikiSym'05*), pp. 75–82, ACM Computer Society, New York, NY, USA, 2005.
- [Formulat] “Formulator, an extensible framework that eases the creation and validation of web forms for Zope”. <http://www.infrae.com/download/Formulator>.
- [Frat99] Piero Fraternali. “Tools and approaches for developing data-intensive Web applications: a survey”. *ACM Computing Surveys*, Vol. 31, No. 3, pp. 227–263, 1999.
- [Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995.
- [Groo] Cees de Groot. “Gardner, a Seaside Wiki”. <http://map.squeak.org/package/6805c4ca-6a33-4396-801a-b7ea1c3e3567>.
- [Grou04] Object Management Group. “Meta Object Facility (MOF) 2.0 Core Final Adopted Specification”. Tech. Rep., Object Management Group, 2004.
- [Grou97] Object Management Group. “Meta Object Facility (MOF) Specification”. Tech. Rep. ad/97-08-14, Object Management Group, Sep. 1997.
- [Inga97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. “Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself”. In: *OOPSLA '97: Proceedings of the 12th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 318–326, ACM Press, Nov. 1997.
- [John98] Ralph Johnson and Bobby Wolf. “Type Object”. In: Robert C. Martin, Dirk Riehle, and Frank Buschmann, Eds., *Pattern Languages of Program Design 3*, Chap. 4, Addison Wesley, 1998. ISBN:0-201-31011-2.
- [Leuf01] Bo Leuf and Ward Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley, 2001.
- [Lien03] Adrian Lienhard. “Mewa: Meta-level Architecture for Generic Web-Application Construction”. Informatikprojekt, University of Bern, Nov. 2003.
- [Mull05a] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. “Weaving Executability into Object-Oriented Meta-Languages”.

- In: S. Kent L. Briand, Ed., *Proceedings of MODELS/UML '2005*, pp. 264–278, Springer, Montego Bay, Jamaica, Oct. 2005.
- [Mull05b] Pierre-Alain Muller, Philippe Studer, Frédérick Fondement, and Jean Bézivin. “Independent Web Application Modeling and Development with Netsilon”. *Software and System Modeling*, Vol. 4, No. 4, pp. 424–442, Nov. 2005.
- [Plus05] Roland Plüss and Philippe Marschall. “Spielverderber, an Access Control List (ACL) based security framework for Pier”. 2005. <http://smallwiki.unibe.ch/advanceddesignlabs/admin/>.
- [Putn] Colin Putney. “OmniBrowser, an extensible browser framework for Smalltalk”. <http://www.wiresong.ca/OmniBrowser>.
- [Reng03] Lukas Renggli. “SmallWiki: Collaborative Content Management”. Informatikprojekt, University of Bern, 2003.
- [Reng06] Lukas Renggli. “Pier Unix Security, an Unix file-system based security framework for Pier”. 2006. <http://map.squeak.org/package/1ae18f4e-086a-46e3-83ff-72ab6673c382>.
- [Reng07] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. “Magritte — A Meta-Driven Approach to Empower Developers and End Users”. In: Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, Eds., *Model Driven Engineering Languages and Systems*, pp. 106–120, Springer, Sep. 2007.
- [Rieh05] Dirk Riehle, Michel Tilman, and Ralph Johnson. “Dynamic Object Model”. In: *Pattern Languages of Program Design 5*, Addison-Wesley, 2005.
- [Riva96] Fred Rivard. “Smalltalk: a Reflective Language”. In: *Proceedings of REFLECTION '96*, pp. 21–38, Apr. 1996.
- [Seaside] “Seaside home page”. <http://www.seaside.st>.
- [Sque10] Squeak. “Squeak Home Page”. <http://www.squeak.org/>, archived at <http://www.webcitation.org/5p1poT9Ta>, 2010.
- [WikiPedi] “WikiPedia, a web-based, free-content encyclopedia”. <http://www.wikipedia.org>.
- [Wool96] Bobby Woolf. “The Null Object Pattern”. In: *Design Patterns, PLoP 1996*, Robert Allerton Park and Conference Center, University of Illinois at Urbana-Champaign, Monticello, Illinois, 1996.

- [Wues] Klaus Wuestefeld. “Prevayler, a prevalence layer for Java”. <http://www.prevayler.org>.
- [Yode01] Joseph Yoder, Federico Balaguer, and Ralph Johnson. “Architecture and Design of Adaptive Object Models”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, pp. 50–60, 2001.
- [Yode02] Joseph W. Yoder and Ralph Johnson. “The Adaptive Object Model Architectural Style”. In: *Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)*, Aug. 2002.