Reflective designs

R. Hirschfeld and R. Lämmel

Abstract: The authors render runtime system adaptations by design-level concepts such that running systems can be adapted and examined at a higher level of abstraction. The overall idea is to express design decisions as applications of design operators to be carried out at runtime. Design operators can implement design patterns for use at runtime. Applications of design operators are made explicit as design elements in the running system such that they can be traced, reconfigured, and made undone. This approach enables reflective designs: on one side, design operators employ reflection to perform runtime adaptations; on the other side, design elements provide an additional reflection protocol to examine and configure performed adaptations. The approach helps understanding the development and the maintenance of the class of software systems that cannot tolerate downtime or frequent shutdown-revise-startup cycles. The authors have designed and implemented a class library for programming with reflective designs in Squeak/Smalltalk. The library employs reflection and dynamic aspect-oriented programming. In addition to that, the authors have implemented tool support for versatile navigation in a system that is adapted continuously at runtime.

1 Introduction

1.1 Problem context: runtime system adaptations

Our work on *ReflectiveDesigns* is concerned with adaptation of software systems at runtime, as needed for dynamic component coordination [1], runtime system configuration [2], dynamic service adaptation [3, 4], and rapid prototyping without shutdown-revise-startup cycles [5]. Runtime adaptability is crucial for systems with strong availability demands; think of telecommunications. Downtime of such systems can barely be tolerated. Software maintenance and evolution have to be carried out in the running system.

ReflectiveDesigns enhance object-oriented design and programming by techniques for runtime system adaptation. There are two key notions: *design elements* and *design operators*, which we will explain in turn.

1.2 Key notion I: design elements

We contend that a program is structured according to design decisions. We require that design decisions are represented explicitly in the program. Thereby, software design becomes traceable in the program. We even require that design decisions are to be represented explicitly in the running system. We use the term design element to denote representations of design decisions in programs. In fact, we require that design elements are amenable to reflection such that design decisions can be observed and modified at runtime. With that, the notion of runtime system adaptations boils down to explicit construction, modification, and retirement of design elements.

1.3 Key notion II: design operators

When compared to basic techniques such as the use of a metaobject protocol [6], the use of design elements makes runtime system adaptations more disciplined and more manageable. To this end, we provide abstractions that capture common design elements in a reusable manner. Applications of such abstractions perform system adaptations at a design level; hence, we call them design operators. Our work, so far, has concentrated on operators that model the realisation of common design patterns [7]. The view 'design patterns as operators' also occurs in previous work on static metaprogramming [8–13]. The novelty of our work is that our operators serve for *runtime* system adaptation, and *runtime* reflection on designs.

1.4 Practical realisation in Squeak/Smalltalk

We have designed and implemented a *ReflectiveDesigns* framework, which is a class library for Squeak/Smalltalk. The implementation makes original use of infrastructure for reflection, method wrappers [14], and aspect-oriented programming with AspectS [15]; [Note 1]. Using the *ReflectiveDesigns* framework, we have exercised some scenarios of runtime system adaptations, as illustrated later.

2 A motivating example

We will now exercise run-system adaptations.
Implementation details are omitted at this stage.

© IEE, 2005

IEE Proceedings online no. 20041097

doi: 10.1049/ip-sen:20041097

Paper first received 13th July and in revised form 19th September 2004

R. Hirschfeld is with DoCoMo Communications Laboratories Europe, Munich, Germany

R. Lämmel is with the Department of Information Management and Software Engineering, Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam, The Netherlands and also with Centrum voor Wiskunde en Informatica, NL-1098 SJ, Amsterdam

Note 1: *AspectS* has been developed and is maintained by the authors. It is available for the Squeak implementation of Smalltalk at http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/. We plan to distribute the *ReflectiveDesigns* framework with *AspectS*.

2.1 Construction of design elements

We use a simple scenario:

> Ulrike and Ellen work in a company that runs a distributed information system to automate most computer-based work. Ellen has observed that Ulrike's printouts are generally of interest as they cover subjects like software maintenance. So the two agree on decorating Ulrike's print service such that each print job that is initiated by Ulrike is also forwarded to Ellen via the email protocol.

This can be accomplished while the information system is up and running, and all relevant objects that have to be adapted already exist. In this example, we do not assume that the availability demands for our information system are exceptionally high. Nevertheless, we assume that our system evolves too frequently for shutdown-revise-startup cycles to be tolerable, which implies that runtime adaptations have to be considered.

The following steps need to be carried out:

1. We construct a design element `deco` for decoration.
2. We configure `deco`'s code block for decoration to forward print jobs to Ellen.
3. We configure `deco`'s target object to be the print service of Ulrike.
4. We activate the design element `deco`.

We use the term decoration here in the sense of the *decorator* design pattern [7]. We recall that the intent of this design pattern is the following: "Attach additional responsibilities to an object dynamically." However, it is important to note that our approach does not require any preparation of the source code. It is not just the decorated functionality that is chosen at runtime. The entire possibility of decoration is added at runtime by using the design operator for decoration. Hence, we have completed a scenario of an unanticipated runtime system adaptation.

2.2 Examination of design elements

We continue the scenario from above:

> The company is reviewed regarding internal security and confidentiality guidelines. As part of the review, the information system is examined. The referees want to systematically browse through the system to assess any redirection, any decoration, and any other modification that is active in the system.

As a result, each active design element is examined as follows:

- Is there undesirable data flow among the 'participants' in the design element?
- Is there any design element that seems to be unused?
- Is the behaviour of adapted methods reasonably preserved?
- Are the reasons for adapted methods reasonably clear?

The introspection interface of design elements allows one to easily navigate to participants, to view code blocks for adaptation, and to assess other meta-data. Hence, the required review task is carried out more easily for design elements than for low-level system patches.

2.3 Reconfiguration of design elements

The above examination implied a corrective system adaptation:

> The referees stated that arbitrary forwarding would incidentally also forward documents that are readily labelled as confidential. Hence, forwarding should be constrained accordingly.

We need to reconfigure the design element `deco` as follows:

1. `deco` is deactivated immediately until the problem is solved.
2. `deco` is reconfigured to perform conditional forwarding. The condition checks whether the confidentiality of the document at hand rules out forwarding.
3. `deco` is reactivated.

It is worth mentioning that a reconfiguration of `deco` is not the only option for the required system adaptation. We could also have added an additional design element that adapts `deco` so that the combined behaviour is conditional forwarding. Yet another option is that `deco` retires altogether, and a better design element is built from scratch. These different options can all be accomplished while the system is running.

2.4 Retirement of design elements

Eventually, we might face the following situation:

> Forwarding is found to be convenient throughout the company. To this end, a general feature is designed such that all system users can subscribe to any employee's print service, while each individual has to regulate whether a document is publicly visible or not.

This scenario is implemented as follows:

1. `deco` is deactivated.
2. `deco` retires.
3. A new, more general design element for forwarding is put into service.

In fact, this general feature for forwarding is eventually accepted as a permanent part of the system. So it is decided that the forwarding feature is woven into the system while offline on Christmas Eve. Making system adaptations permanent like that eliminates runtime overhead of transient adaptations.

3 Basics of system adaptations

Before we lay out the advanced notions of design elements and design operators, we will review relevant basics of system adaptations. In particular, we will discuss the different intents of adaptability, and the different times when to adapt systems. Eventually, we will discuss two major options for runtime system adaptations:

- Adding variation points for anticipated adaptations to the system design.
- Employing language support for unanticipated adaptations.

We will relate to standard design problems in illustrative examples.

3.1 Hard-coded variations

In Fig. 1, there is a class hierarchy that involves three different traversal operations. Parts of the traversal code are scattered over the classes in the class hierarchy. Adding yet another traversal operation in the same way would imply changing all classes.

In Fig. 2, there is an extended `StringMorph` class from Squeak's Morphic system to allow for bordered string morphs. The variation bordered vs. unbordered string morphs is expressed here by conditional code. Adding

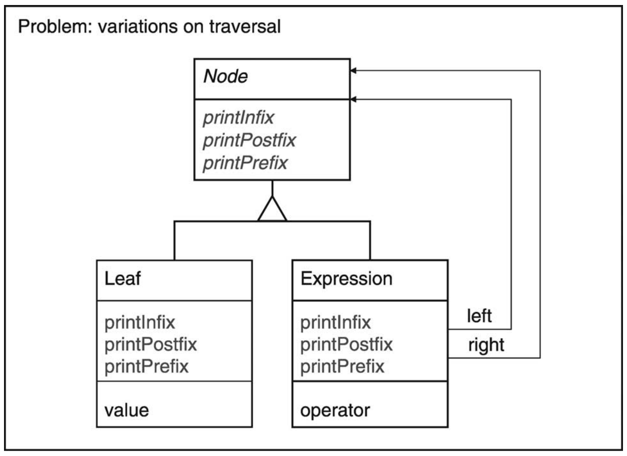


Fig. 1 *Traversal code scattered over a class hierarchy*

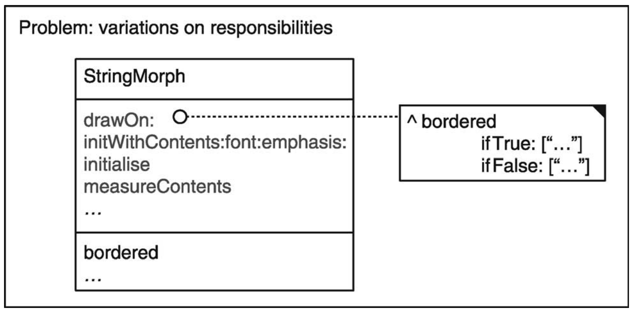


Fig. 2 *Conditional code for variant selection*

another variation would imply changing the actual class `StringMorph`.

Both designs do not make variation points explicit in the design, but they rather hardcode a fixed number of variations. New variations cannot be isolated in a modular manner, but existing code would have to be modified. Consequently, the development-time extensibility of these designs is poor.

3.2 Variation points in the system design

The aforementioned circumstances call for *refactoring* such that the design patterns *visitor* and *decorator* are instantiated. The resulting designs are shown in Fig. 3 and Fig. 4. As for the visitor (Fig. 3), we implement an interface for the double-dispatch protocol in each class of the `Node` hierarchy that is subject to traversal. Then, each new traversal can be defined as a subclass of *NodeVisitor*, which models a family of `visit` methods — one for each kind of object. As for the decorator (Fig. 4), we use object composition to equip `StringMorphs` with additional behaviour for borders.

We note that refactoring, with its inherent impact on code, is normally understood as a form of system adaptation that is done during development and maintenance time [16, 17], when the system is offline. There is no fundamental reason to restrict refactoring to offline transformations. However, corresponding runtime approaches require sophisticated techniques to deal with the migration of objects that instantiate affected classes [18–21].

3.3 Adaptability intents

The two refactored designs both expose variation points that were missing in the original designs, and that were needed for variation in certain dimensions. However, the two examples emphasise different intents of adaptability:

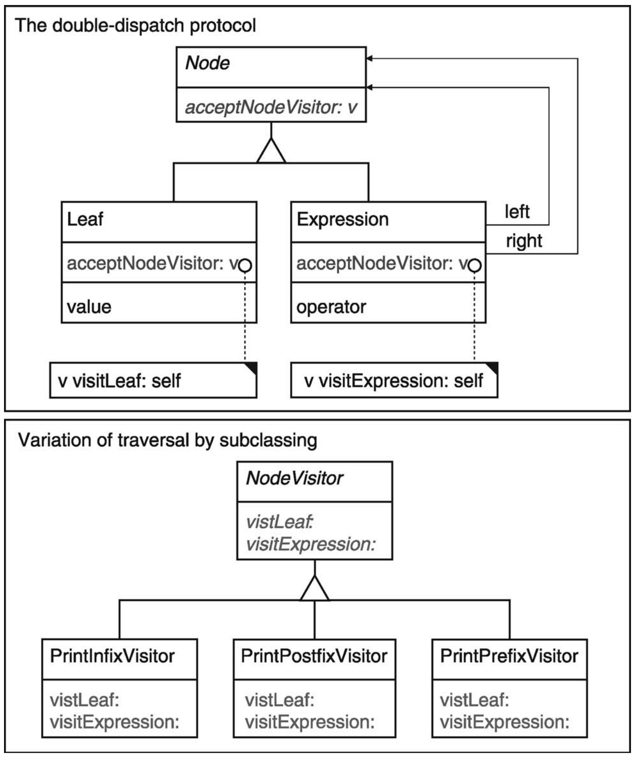


Fig. 3 *A visitor for modularised traversal operations*

- Development-time modularity (*visitor*).
- Runtime adaptability (*decorator*).

That is, the primary promise of the *visitor* pattern is to allow for adding new traversals by means of providing just one new class for each traversal, without changing existing classes. This is clearly an example of modular extension at development time. It is merely a convenient side effect that such a design also provides benefits at runtime:

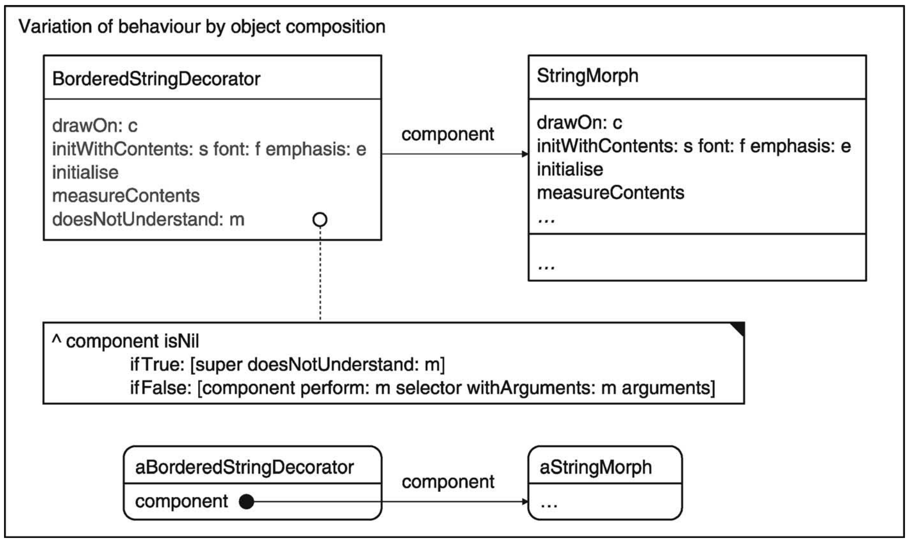
- Runtime extensibility: a new traversal class can be loaded dynamically.
- Runtime adaptability: functionality can be parameterised in visitor objects.

By contrast, the issue of runtime adaptability is dominating for the *decorator* pattern. The primary promise of the pattern is to allow for dynamic attachment of responsibilities by means of plain object composition. By committing to this pattern at development time, we anticipate the corresponding kind of runtime adaptations.

3.4 Limits of anticipated runtime adaptations

The conventional implementation of the *decorator* pattern implies that the decorated object and the undecorated object carry different object identities. This often leads to 'object schizophrenia': upon completed decoration, parts of the program might accidentally continue to refer to the undecorated object because this is the reference that was stored initially. There are several ways of tackling this problem, e.g.:

- *Obligatory top decorator*. The design can be elaborated as follows. Objects, that are potentially subject to decoration, are hosted by a dedicated top decorator. Elsewhere in the system, we always refer to the top decorator. Any additional decorator ends up between top decorator and hosted object.
- *Low-level redirection*. If available, we can employ an (expensive) operation to replace references to the undecorated object by the reference to the decorated object. For instance, Squeak provides an operation `become:with:`.



Variation of behaviour by object composition
BorderedStringDecorator
drawOn: c
initWithContents: s font: f emphasis: e
initialise
measureContents
doesNotUnderstand: m
component
StringMorph
drawOn: c
initWithContents: s font: f emphasis: e
initialise
measureContents
...
...
^ component isNil
ifTrue: [super doesNotUnderstand: m]
ifFalse: [component perform: m selector withArguments: m arguments]
aBorderedStringDecorator
component
component
aStringMorph
...

Fig. 4 *A decorator for dynamic attachment of responsibilities*

Both solutions indicate general limitations of anticipated runtime adaptations. The first solution illustrates the tension between complexity of designs and robustness of adaptation techniques. The second solution demonstrates incomplete anticipation.

3.5 *Unanticipated runtime adaptations*

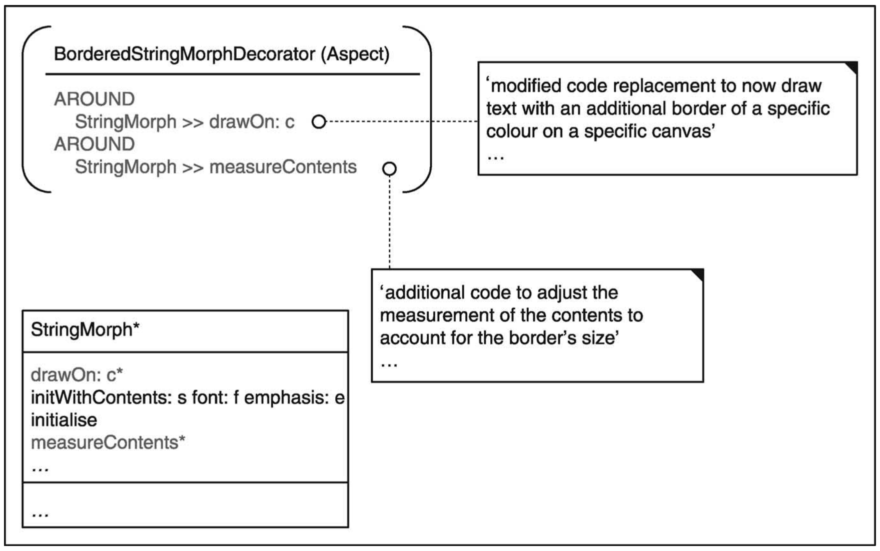
So far we have focused on the classic OOD approach. That is, we have anticipated adaptations in the design. A different approach is to employ language support for unanticipated adaptations. In Fig. 5, we approach to the decoration problem by employing an advanced modularisation technique. That is, we employ runtime *around advice* for methods, as provided by *AspectS* [15] and other frameworks for dynamic aspect-oriented programming. Such expressiveness is also called method-call interception elsewhere [22, 23]. Consequently, the design in Fig. 5 does not just comprise normal classes but also an aspect, which hosts the around advise for the two methods `drawOn` and `measureContents` that need to be adapted for bordered `StringMorphs`. Variation of behaviour does not need to be anticipated in the static design any longer because the aspect can be dynamically woven. The aforementioned problem with object schizophrenia does not appear because there is just one object reference.

3.6 *Runtime provision of variation points*

While the *visitor* pattern provided us with development-time modularity for traversals, we can further improve on this design by employing a runtime adaptation. In Fig. 6, we use runtime *introductions* to superimpose the double-dispatch protocol of the *visitor* pattern onto the system. That is, the aspect adds implementations of the method `acceptNodeVisitor` for the two concrete classes `Leaf` and `Expression` that are involved in the `Node` hierarchy. Such expressiveness is again readily provided by *AspectS*. This example illustrates once more that runtime adaptations can be used to add variation points that otherwise had to be anticipated at development time.

3.7 *Summary*

This Section went through a transition from variation points that are explicit in the design of a system to unanticipated adaptations based on dedicated language support. There are different forms of such language support:



BorderedStringMorphDecorator (Aspect)
AROUND
StringMorph >> drawOn: c
AROUND
StringMorph >> measureContents
'modified code replacement to now draw text with an additional border of a specific colour on a specific canvas'
...
'additional code to adjust the measurement of the contents to account for the border's size'
...
StringMorph*
drawOn: c*
initWithContents: s font: f emphasis: e
initialise
measureContents*
...
...

Fig. 5 *Aspectual decoration with around advice*

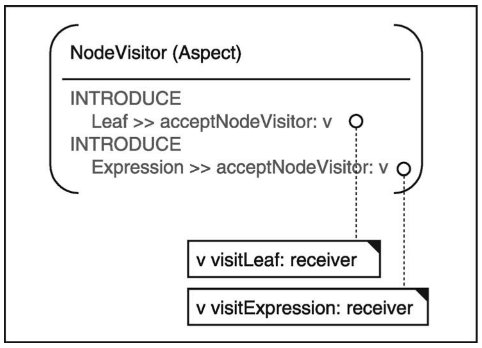

NodeVisitor (Aspect)
INTRODUCE
Leaf >> acceptNodeVisitor: v
INTRODUCE
Expression >> acceptNodeVisitor: v
v visitLeaf: receiver
v visitExpression: receiver

Fig. 6 *Aspectual introduction of double-dispatch protocol*

• Primitives for reference replacement, dynamic class loading, and others.
• Superimposition of structure and behaviour onto a system based on a join-point model as in aspect-oriented programming [24, 25]. We note that static weaving is sufficient for modular extension at the code level, while dynamic weaving is necessary for runtime system extension.
• A metaobject protocol or meta-programming framework can allow for almost arbitrary rewriting of the system. Rewriting can be performed at different times including compile-time and runtime.

4 Requirements for reflective designs

Reflective designs are designs that employ design elements and design operators. We will now characterise reflective designs in more details.

4.1 Design elements — explicit representations of design decisions

We require that design decisions are represented explicitly at the source code level and in the running program. We assume that design decisions can be viewed as runtime system adaptations. We use the term 'design element' to denote representations of design decisions in the running system. Technically, a design element can be modelled as follows:

- We use meta-data to represent the design decisions.
- We use a proper object instead.
- We use a dedicated team of objects.

We summarise the life cycle of design elements in Fig. 7. A design element can be in three states: inactive, active, or retired. We say that a design element is inactive when it does not affect the system. We assume that design elements are inactive upon construction. Upon completed configuration, they can be activated. Retirement of active design elements requires previous deactivation.

4.2 Design operators — parameterised abstractions for design elements

We obviously need a scheme for making design decisions operational. To this end, our approach to reflective designs provides operators for runtime system adaptations. It is clear that such operators can only be provided in the context of a sufficiently reflective programming system. We classify design operators as follows:

• *Additive operators* superimpose additional structure or behaviour onto the running software system. These operators rely on a join-point model to refer to points in the execution of the given system, just as in aspect-oriented programming. The applications of additive operators, like all design operators, can be made undone.
• *Subtractive operators* identify and remove slices of behaviour or structure in the running software system. We note that subtractive operators are needed to remove parts of a system that were statically contributed. (That is, subtractive operators are not needed to reverse the effects of additive operators, since the latter is merely about making adaptations undone.) Subtractive operators, like additive operators, rely on a join-point model.
• *Refactoring operators* revise the running system in a semantics-preserving manner. Such adaptations can involve invasive code changes, changed interfaces, and changes in terms of the join points that are exposed by the system.

Actual applications of such operators result in two effects. Firstly, the corresponding design elements are constructed. Secondly, the system's actual structure and behaviour is adapted as intended by the design decision at hand. Applications of design operators are undone by deactivating the corresponding design element. In case an inactive element is never ever needed again, we can let the element retire. In addition to the classification given above, we expect design operators to be grouped in categories. For example, different implementations of the same design pattern are grouped in a category.

4.3 Design-level reflection

Design elements can be examined and (re-) configured. Examination and (re-) configuration are meant in the sense of introspection and intercession. We recall that introspection denotes the observation or read-only part of reflection, while intercession denotes the modification or write-enabled part of reflection. Classic object-oriented introspection and intercession concerns the fields and methods of objects. Design-level introspection and intercession are concerned with design-level concepts such as the list of *participants* of a given design element. A single participant is simply a pair of a descriptive role and an entity of the following kind:

- A class or a set thereof.
- An object or a set thereof.
- The above two but qualified with method or field names.

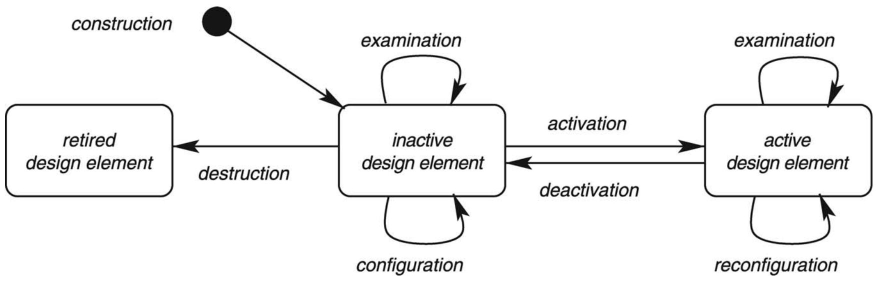


Fig. 7 *The life cycle of design elements*

For instance, a design element for a decorator lists participants as follows. There is a role *component*, which is mapped to an object qualified by a method name. There is also a role *decorator*, which is mapped to an object with the additional responsibility. A slightly more general scheme for a decorator is that the role *component* maps to a set of objects qualified by a method name. This allows for reuse of decorator instances.

Participants can be examined for each design element. This is design-level introspection. Design elements can be configured with regard to specific participants for prescribed roles. This is design-level intercession. Furthermore, for each object in the running system, we can introspect the *adaptations*, i.e., a list of design elements, that affect the object at hand.

5 Programming with reflective designs

We will now illustrate the use of the *ReflectiveDesigns* framework from the perspective of a Squeak/Smalltalk programmer. To this end, we will trivialise the decoration example from the motivation section so that we can actually exercise the life cycle of a design element in just a few lines. We will consider a demo scenario for a smart proxy.

5.1 Objects to deal with

We assume an object `myObj` that will eventually serve as the *subject* of the proxy in the terminology for the *proxy* pattern [7]. This object is of class `MySampleClass`, which implements methods as follows:

```
returnSeven
  ↑7  "returns integer 7"
returnThree
  ↑3  "returns integer 3"
```

We also assume an object `myRealSubj` that will eventually serve as the *real subject* of the proxy, again, according to the common terminology. That is, `myRealSubj` provides new behaviour meant to replace the one of `myObj`. The object `myRealSubj` defines the methods `returnSeven` and `returnThree` differently:

```
returnSeven
  ↑ 'seven' "returns string seven rather
             than integer 7"
returnThree
  ↑ 'three' "returns string three rather
             than integer 3"
```

5.2 Construction, configuration, activation

All the statements that follow can be executed just in the given order. The following line constructs the proxy:

```
myProxy ← RdProxyAspect new.
```

The class `RdProxyAspect` provides a specific aspect-oriented implementation of the *proxy* pattern, but these implementation details are not relevant here. We only need to obey the configuration protocol for proxies. The following lines of code start with the configuration of `myProxy`:

```
myProxy
  proxy: MySampleClass
  selectors:{#returnSeven} asSet.
```

This configuration captures that the proxy is supposed to affect objects of class `MySampleClass`, and more specifically invocations of the method `returnSeven`. This configuration is sufficient for the moment. We activate the smart proxy as follows:

```
myProxy activate. "system adaptation
                   becomes effective"
```

5.3 Impact tracking and reconfiguration

We note that proxies according to `RdProxyAspect` are instance-specific in the sense that each affected subject and the corresponding real subject need to be registered explicitly. Hence, the configuration, performed so far, is incomplete; we only have specified the class of affected objects so far. This incomplete configuration manifests itself such that all methods of `myObj` still compute as usual. We capture this expectation in assertions as follows:

```
myObj assert: myObj returnThree = 3.
  "still returns 3"
myObj assert: myObj returnSeven = 7.
  "still returns 7"
```

These assertions are executed silently, which means that our assumptions about the system behaviour are substantiated. The following lines reconfigure the proxy such that it has an impact on the system:

```
myProxy
  addSubject: myObj    "what to intercept"
  realSubject: myRealSubj. "what to do"
```

That is, we configure `myProxy` such that `myRealSubj` provides the new functionality, and `myObj` is going to be affected. The invocation of `returnSeven` should give a different result than before, while the invocation of `returnThree` should still behave the same. We can again capture these expectations in assertions as follows:

```
myObj assert: myObj returnThree = 3.
  "unchanged"
myObj assert: myObj returnSeven = 'seven'.
  "adapted!!"
```

The assertions demonstrate the impact of the activated design element `myProxy`.

5.4 Retirement of the design element

The following statements illustrate the retirement of the proxy:

```
myProxy deactivate. "adaptation no longer
   appreciated"
myObj assert: myObj returnThree = 3.
  "as before"
myObj assert: myObj returnSeven = 7.
  "back to normal"
```

This style of exercising the life cycle of design elements, including the use of assertions, qualifies the session as a useful unit test. In the *ReflectiveDesigns* framework, we document design operators and adaptation scenarios just like that.

6 An architecture for reflective designs

We have implemented reflective designs as a class library in Squeak/Smalltalk — the *ReflectiveDesigns* framework. We will now discuss the overall structure of the *Reflective-Designs* framework. We will report on essential programming techniques used in the implementation of the framework, and will link these techniques to achievable benefits such as reconfigurability and adaptiveness. Finally, we will assess the transposition of typical design patterns to the level of reflective designs.

6.1 Housekeeping

Our current implementation of the *ReflectiveDesigns* framework comprises a central authority — the *ReflectiveDesigns* center, which is a special object `RdCenter`. All design operators and design elements are known to `RdCenter`. The registration and management of all these entities rely on reflection and existing Squeak/Smalltalk protocols. For instance, each design element registers with `RdCenter` upon construction as part of the initialisation phase:

```
initialize
  self addDependent: RdCenter current.
```

That is, we use Smalltalk's change/update protocol to establish a link between design elements and `RdCenter`. Generic design-level introspective functionality is supported by the API of `RdCenter`. For instance, the following services are provided:

- `designOperators` — all classes for design operators.
- `designOperators: aCategory` — dito, but per category.
- `designCategories` — all such categories.
- `designElements` — all active and inactive design elements.
- `designElements: anOperator` — dito, but per operator.

As explained earlier, each object can be introspected to report the `adaptations` applied to it. For reasons of efficiency, `RdCenter` also contributes central data structures for such object-based introspective services. Likewise, the `RdCenter` contributes to additional tests for design elements, e.g.:

- `affects: anObject` — test if the design element affects the given object.
- `affects: anObject selector: aSymbol` — dito, but per method.

6.2 Programming layers

The *ReflectiveDesigns* framework involves the layers shown in Fig. 8. The top layer is for the user of reflective designs. This layer hosts classes for *design operators*. The functionality implemented by these classes supports the life cycle of design elements. The next layer, *adaptation idioms*, provides functionality and data structures for traceable runtime system adaptations. This layer promotes reuse in the top layer. The next layer, *AspectS*, is the aspect-oriented framework *AspectS* [15], which can be conveniently used to describe some typical kinds of system adaptations. The next layer, *method wrappers* [14], is employed in the implementation of *AspectS*. The bottom layer stands for low-level reflection via the metaobject protocol (MOP) of Squeak/Smalltalk. The idea is that layers at a higher level of abstraction perform less lower level reflection.

6.3 Aspect-oriented idioms

Some forms of additive and subtractive adaptations can be conveniently expressed by concepts of aspect-oriented programming (AOP), as available in the dynamic AOP framework *AspectS*. In particular, the following kinds of adaptations can be expressed:

- Introduction of methods.
- Removal of methods.
- Revision of method behaviour.

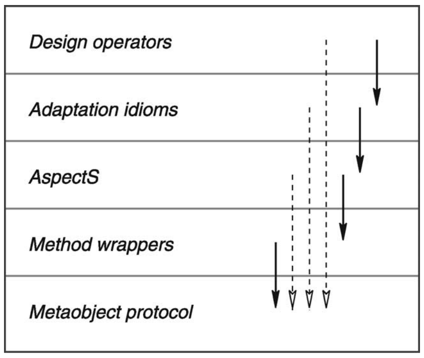


Fig. 8 *Layers in the ReflectiveDesigns framework*
The top layer provides the operators for the framework user. The bottom layer provides low-level reflection for reflective designs. The inner layers support runtime system adaptations at increasing levels of abstraction

One can use pointcuts to address the places in a system that are to be adapted. Advice can be used to define new methods, which is also called a runtime introduction, or to revise existing methods, which is also called method-call interception.

There are two idioms that help us to detach ourselves from plain AOP at the top layer for design operators. We place these idioms in a class `RdAspect` (read as *ReflectiveDesigns* — aspects), which inherits from *AspectS*' base class `AsAspect`:

- Introduce a method into a given class.
```
introduce: aClass
  selector: aSymbol
  with: aBlockContext
  qualifier: anAsAdviceQualifier
```
- Perform method-call interception for a method of a given class.
```
intercept: aClass
  selector: aSymbol
  with: aBlockContext
  qualifier: anAsAdviceQualifier
```

The implementation of this API supports traceability and participation in change/update protocols. These idioms are particularly convenient for 'point-wise' system adaptations, where a specific object or a specific method is adapted.

It is instructive to notice the following link between classic OOD and *ReflectiveDesigns*. While classic object-oriented designs anticipate variation points via the use of subclassing and object composition, reflective designs additionally employ the two dynamic AOP idioms described above.

6.4 Reconfigurability by data dictionaries

Design operators often deal with mapping entities such as objects or methods to other entities of the same kind or simply to code bocks. This can be viewed as drawing 'connectors' in an architectural sense. With regard to design patterns, a good example is the *mediator* pattern [7], where colleagues request services, and these requests are handled by the mediator such that they are forwarded to suitable colleagues. In the *ReflectiveDesigns* framework, we employ data dictionaries in the implementation of all design operators that draw connectors. Data dictionaries provide a high degree of reconfigurability because they can represent connectors very explicitly. By contrast, a conventional implementation of connectors limits traceability and reconfigurability. For example, the *mediator* pattern [7] is normally implemented by statements that translate incoming calls (somehow) into outgoing calls. Hence, the connectors are not immediately traceable.

6.5 Programming techniques at a glance

As a kind of benchmark, we have investigated the design patterns of 'Gang of Four' (GoF; [7]) regarding the possibility of using a number of programming techniques in their implementations. In this investigation, we have found attractive implementations that serve runtime system adaptations better than conventional approaches. In Fig. 9, we show all GoF patterns with columns for selected programming techniques. We retain the normal grouping of creational, structural and behavioural patterns.

Programming techniques covered in Fig. 9 are:

- The use of data dictionaries as described above.
- The use of structural reflection, beneficial for making designs more adaptive or generic. A good example is the *visitor* pattern where the sub-objects of any given object can be determined by introspection.
- The use of method-call interception as described above.
- The use of runtime introductions as described above.

We summarise: pattern implementations that involve data dictionaries, runtime introductions, method-call interception, or structural reflections are meaningful examples of design operators. Such added value can be reported for 16 out of 23 patterns. For the remaining 7 patterns, we did not find runtime solutions with added value. This either means that the conventional implementation is satisfactory, or an implementation is trivial simply because the pattern is readily supported in Squeak/Smalltalk — as in the case of iterators. We note that a minor benefit can be claimed for all 23 design patterns. That is, the mere construction of design elements, at the very least, serves the traceability (or documentation) of design decisions.

Pattern	Data dict.	Reflection	Intercept.	Introduct.
Abstract Factory	*			*
Builder	*	*		*
Factory Method				
Prototype	*	*		
Singleton				
Adapter	*			*
Bridge	*		*	
Composite				
Decorator	*		*	
Facade				
Flyweight				
Proxy	*		*	
Chain of Resp.			*	
Command				
Interpreter				
Iterator				
Mediator	*			
Memento		*		
Observer	*		*	
State	*		*	
Strategy	*		*	
Template Method			*	
Visitor	*	*		*

Fig. 9 *Techniques used in ReflectiveDesigns implementations of design patterns*

7 Implementations of design operators

We will now discuss representative implementation details for some design operators The selected operators implement the design patterns *visitor*, *factory*, and *proxy* in a specific manner. We will demonstrate the role of programming techniques for basic reflection, dynamic weaving, and data dictionaries.

7.1 Implementation of the visitor pattern

In Section 3, we outlined a runtime version of the *visitor* pattern, where a double-dispatch protocol is injected into the system using runtime introductions. Another possible implementation of the *visitor* pattern is to modularise all `visit` methods in one aspect, and then to inject these various methods into the corresponding classes, again, by runtime introductions. The double-dispatch protocol is not needed for this option. We will now consider yet another implementation, which is highly generic and flexible.

Our generic visitor object implements the following `genericVisit` method:

```
genericVisit: anObject
  (self visited includes: anObject)
  ifFalse: [
    self count: self count + 1.
    anObject ifNotNil: [self visited add:
      anObject].
    (self traversalStop: anObject)
      ifFalse: [
        self depth: self depth + 1.
        (self traversalStrategy: anObject)
           value: anObject.
        self depth: self depth− 1]].
```

Given `anObject,` the method first checks whether this object has been `visited` already. We only continue if this test evaluates to `false,` since we want to rule out cycles. We continue by incrementing a node counter, and by adding the object to `visited.` Then we check whether the object is registered in `traversalStop.` We only continue if this test evaluates to `false.` Next we lookup and invoke the specific `traversalStrategy` for `anObject,` while we maintain an increased `depth` during this step. We refer to [26] for a discussion of generic visitors in Smalltalk, which however does not explore all the variation points considered here.

The method `traversalStrategy` immediately accesses a data dictionary that maps classes to blocks with class-specific traversal strategies. The use of a data dictionary makes visitors highly configurable. The employment of Smalltalk's metaobject protocol is made explicit in the following default for `traversalStrategy:`

```
defaultIndexedInstVarTraversalStrategy
↑ [:anObject |
1 to: anObject class instSize do: [:idx |
  (anObject instVarAt: idx) genericAccept:
     self]]
```

That is, we iterate over instance variables of the given object, and we let them `accept` the visitor at hand In fact, we assume a method `genericAccept`, to be implemented by any object that is encountered during traversal. As a simple example of a design operator, we consider the injection of this `genericAccept` method into the system. The corresponding operator `RdGenericAccept` subclasses *AspectS*'s base class `AsAspect` because the underlying adaptation will employ runtime introductions. The system adaptation is described by the following method:

```
introGenericAcccept
  self
    introduce: Object
    selector: #acceptGenericVisitor :
    with:  self genericAcceptBlock
    qualifier: {#receiverClassSpecific }.
genericAcceptBlock
    [:receiver:arguments:aspect:client |
  arguments first genericVisit: receiver].
```

So the generic accept method is added to the class `Object`. Its implementation immediately redirects the `genericAccept` method to the `genericVisit` method, but subclasses can favour a different scheme as a means of customising traversal.

7.2 Implementation of the factory pattern

A factory is an object that is able to construct 'products' of different kinds. The conventional approach [7] is to handle each product by a dedicated method. Another approach is to use a data dictionary as a mapping from product descriptors to code blocks for product construction. The following implementation combines these two options. In this way, we obtain a highly configurable factory that is still conveniently used via method calls.

The class `RdFactoryAspect` hosts the design operator for factories. The class must subclass `AsAspect` because the underlying adaptation will employ runtime introductions. The system adaptation is described by the following method:

```
extendFactory: aSymbol part:
   aBlockContext classes: aSet
  self
    introduce: self class
    selector:  aSymbol
    with:      aBlockContext
    qualifier:  anAsAdviceQualifier.
  self factoryKeys add: aSymbol.
  self factoryResults at: aSymbol put:
     aSet.
```

This method registers a new product descriptor `aSymbol` together with a block for construction. The descriptor is viewed as a method name, too, and the corresponding method for product construction is injected in the system with a runtime introduction. The method `extendFactory` is also parameterised by `aSet` of classes, which is meant to document the possible classes of the constructed product. The product descriptions and the intended classes are also maintained in collections `factoryKeys` and `factoryResults`.

As part of the protocol for design-level reflection, we have required that we can access the participants for every design element. We illustrate this concept via the simplified implementation of the `participants` method for the *factory* pattern:

```
participants
  ↑ IdentityDictionary new
    at: #AbstractFactory put: self;
    at: #ConcreteFactory put: self;
    at:#AbstractProduct put: self
       factoryKeys;
    at: #ConcreteProduct put: self
        factoryResults;
    at: #Client put: (Smalltalk pointersTo:
        self class);
    yourself
```

This dictionary relates to the common terminology [7]. There are roles for the abstract and concrete factory, for abstract and concrete products, and their are presumably clients. In the given implementation of the *abstract factory* pattern, the design object itself (namely, `self`) serves as both abstract and concrete factory. The classes of abstract and concrete products are retrieved from the data dictionaries `factoryKeys` and `factoryResults`.

7.3 Implementation of the proxy pattern

The corresponding design operator is hosted in a class `RdProxyAspect`, which subclasses `AsAspect` because the underlying adaptation will employ method-call interception. The method for configuring the proxy is the following:

```
proxy: aClass selector: aSymbol
  self
    intercept: aClass
    selector: aSymbol
    with: (self proxyBlock: aSymbol)
    qualifier: {#receiverInstanceSpecific}.
proxyBlock: aSymbol
  ↑[:receiver :arguments :aspect :client
     :clientMethod |
   (self subjects at: receiver)
     perform: aSymbol
     withArguments: arguments]
```

That is, constructing a proxy comes down to the injection of around advice into the running system. The code block `proxyBlock` forwards intercepted method calls according to the content of a data dictionary `subjects`. Initially, this data dictionary models the identity function.

The following method of `RdProxyAspect` facilitates configuration of the proxy. That is, we can add a subject together with the corresponding real subject:

```
addSubject: anObject realSubject:
   anotherObject
  self receivers add: anObject.
  self subjects add: anObject ->
     anotherObject.
  self changed: {#reflectiveDesigns.self.}.
```

The collection `receivers` keeps track of affected receivers for efficient impact tracking. The collection `subjects` encodes the actual mapping of subjects to real subjects. The last line in the method implementation contributes to the change/update protocol for design elements. That is, a change related to `reflectiveDesigns` is indicated, and `self` is included as a means to report the changed object.

We use the *proxy* pattern to illustrate another point, namely the provision of meta-data and documentation for design operators. In our implementation, classes for design operators are tagged by a method `designOperator`. This allows us to gather all design operators via introspection. For our proxy implementation we have:

```
designOperator
  ↑ RdProxy
```

The return value `RdProxy` describes the category of the operator at hand. In fact, there can be several implementations for the same kind of adaptation scenario. There is also a documentation for each design operator available. For the proxy implementation at hand, there is the following documentation:

```
implementationDescription
↑'This is an implementation of PROXY (207,',
   self designOperator printString,
   '), based on AspectS. ...'
```

8 Tool support for reflective designs

Smalltalk environments are highly interactive. For instance, several ways of browsing the system are supported. Hence, it is important to provide seamless interactive tool support for reflective designs. Accordingly, we have extended some existing tools, and we have provided new tools. The tool extensions are particularly interesting in so far that we have implemented them as self-applications of the *ReflectiveDesigns* framework, e.g., the browsers are adapted by using appropriate design elements.

8.1 Elaboration of browsers and inspectors

Important interactive tools in the Squeak environment are the following:

- System browser: browse all classes as organised in categories.
- Hierarchy browser: browse all classes as organised in the class hierarchy.
- Inspector and object explorer: inspect and change objects.
- Debugger: debug system execution.

Navigating in the system is simplified by connecting these tools and views. For instance, the pop-up menu for a class allows one to inspect all instances of that class. The elaborated systems browser is illustrated in Fig. 10. The elaborated inspector is illustrated in Fig. 11. The elaborations highlight adaptations, and they support new forms of navigation such as

- going from an affected class to the relevant adaptation, or
- going from a design element to any of its participants.

Such design navigation adds to the normal dimensions of navigating in a Smalltalk image. One can readily obtain information about the design-level structure of a system, about previously performed adaptations. Clearly, these navigation features rely on the examination protocol for design elements.

8.2 The ReflectiveDesigns browser

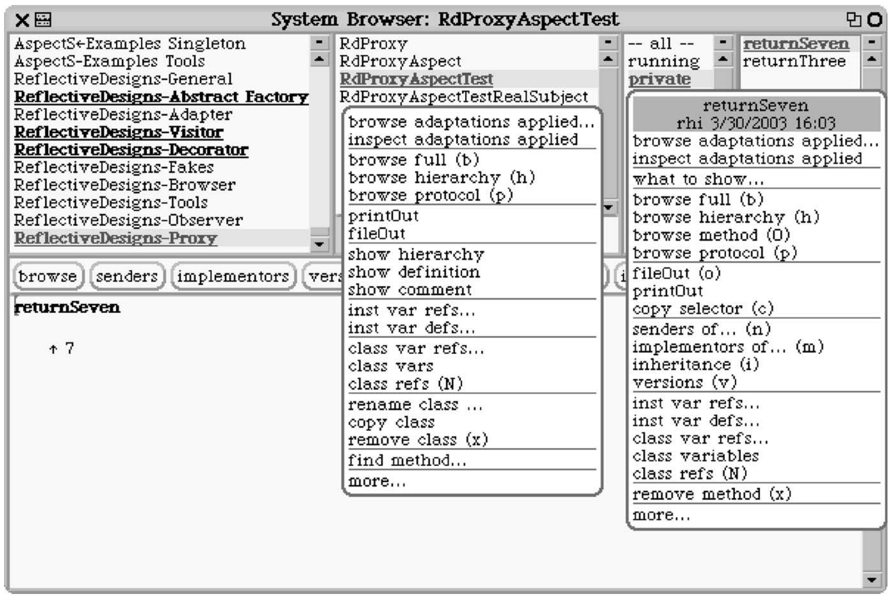
We have developed a new browser, which we call the *ReflectiveDesigns* browser. It bundles several views that relate to design operators and design elements. The *ReflectiveDesigns* browser operates on the *ReflectiveDesigns* center. From the perspective of the *ReflectiveDesigns* browser, the *ReflectiveDesigns* center is a simple, object-oriented framework that views design operators as plug-ins, and that keeps track of inactive and active design elements.

A screen-shot of *ReflectiveDesigns* browser is shown in Fig. 12. There are six panels:

- Left upper panel: registered categories of design operators.
- Left lower panel: documentation for selection in left upper panel.
- Middle upper panel: registered design operators of the selected category.
- Middle lower panel: documentation for selection in middle upper panel.
- Right upper panel: inactive and active design elements of that kind.
- Right lower panel: object explorer for the selected design element.

The pop-menus offer services like the following:

- Browse the documentation of a design operator or its category.
- Browse all references to an operator in the system.
- Inspect the participants of a design element.
- Browse classes and instances that are affected by a design element.

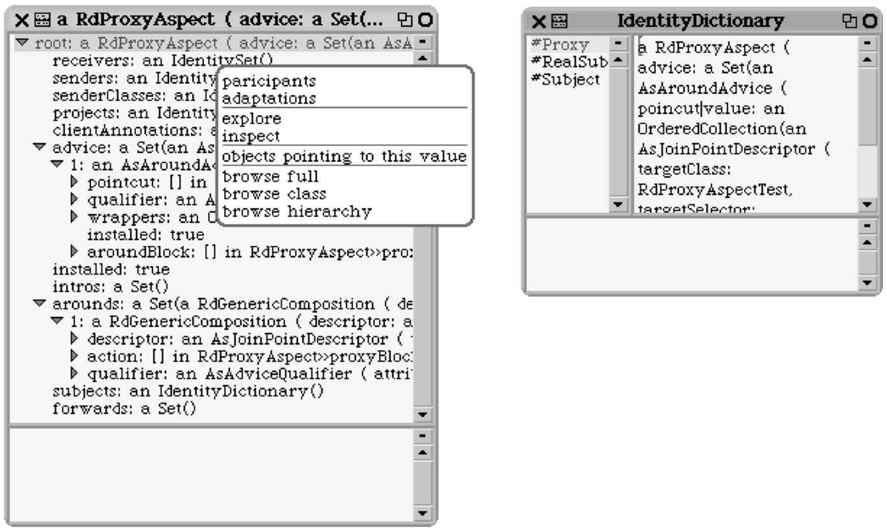


```
returnSeven

    ↑ 7
```

Fig. 10 *Squeak's system browser with awareness of reflective designs*

All classes that are affected by system adaptations are highlighted. The pop-up menus for classes and methods readily allow the programmer to navigate to the "applied adaptations". One can browse the code for the adaptations (i.e., the classes with design operators), and one can inspect the instances for the adaptations (i.e., the design elements)

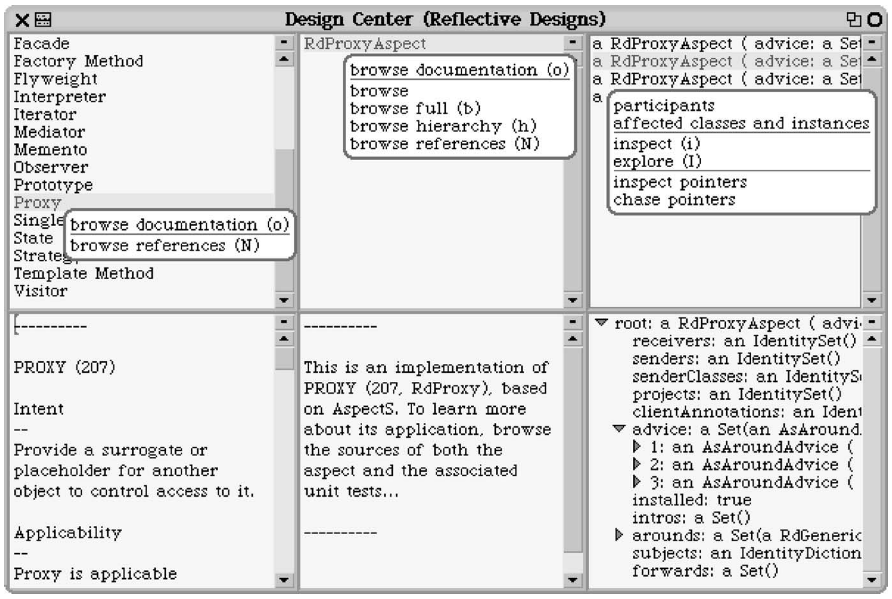


```
a RdProxyAspect ( advice: a Set(...
▼ root: a RdProxyAspect ( advice: a Set(an AsA
    receivers: an IdentitySet()
    senders: an Identity
    senderClasses: an Id
    projects: an Identity
    clientAnnotations: a
  ▼ advice: a Set(an As
    ▼ 1: an AsAroundA
      ▶ pointcut: [] in
      ▶ qualifier: an A
      ▶ wrappers: an O
        installed: true
      ▶ aroundBlock: [] in RdProxyAspect>>pro:
    installed: true
    intros: a Set()
  ▼ arounds: a Set(a RdGenericComposition ( de
    ▼ 1: a RdGenericComposition ( descriptor: a
      ▶ descriptor: an AsJoinPointDescriptor (
      ▶ action: [] in RdProxyAspect>>proxyBloc
      ▶ qualifier: an AsAdviceQualifier ( attri
    subjects: an IdentityDictionary()
    forwards: a Set()
```

```
IdentityDictionary
#Proxy       a RdProxyAspect (
#RealSub     advice: a Set(an
#Subject     AsAroundAdvice (
             poincut value: an
             OrderedCollection(an
             AsJoinPointDescriptor (
             targetClass:
             RdProxyAspectTest,
             targetSelector:
```

Fig. 11 *Inspection of design elements*

We introspect a proxy, which is an instance `RdProxyAspect`; so it is implemented as an aspect. There are fields like `receivers` and `senders` related to the join-point model; proxies can be potentially receiver-specific. There is a compound `advice` field, which is supposed to hold the code block for the proxy. There is also a field telling us that the proxy is `installed`. Via the pop-up menu, we can inspect involved participants in case the object is a design element, and we can inspect applied adaptations for the object, if any. The participants for the proxy are shown on the right



```
Facade
Factory Method
Flyweight
Interpreter
Iterator
Mediator
Memento
Observer
Prototype
Proxy
Single
State
Strateg
Template Method
Visitor
```

```
RdProxyAspect
```

```
a RdProxyAspect ( advice: a Se
a RdProxyAspect ( advice: a Se
a RdProxyAspect ( advice: a Se
a
```

```
----------

PROXY (207)

Intent
--
Provide a surrogate or
placeholder for another
object to control access to it.

Applicability
--
Proxy is applicable
```

```
----------

This is an implementation of
PROXY (207, RdProxy), based
on AspectS. To learn more
about its application, browse
the sources of both the
aspect and the associated
unit tests...

----------
```

```
▼ root: a RdProxyAspect ( advi
    receivers: an IdentitySet()
    senders: an IdentitySet()
    senderClasses: an IdentityS
    projects: an IdentitySet()
    clientAnnotations: an Ident
  ▼ advice: a Set(an AsAround
    ▶ 1: an AsAroundAdvice (
    ▶ 2: an AsAroundAdvice (
    ▶ 3: an AsAroundAdvice (
    installed: true
    intros: a Set()
  ▶ arounds: a Set(a RdGeneric
    subjects: an IdentityDiction
    forwards: a Set()
```

Fig. 12 *The ReflectiveDesigns browser for design operators and design elements*

The categories of design operators all deal with design patterns. The specific design operators in the middle are the implementations that are available for the selected category. The design elements for the selected implementations are shown on the right

With the help of the panels, one can learn about design operators and their uses in the system. More generally, these services enable navigation in the Squeak/Smalltalk image in relation to reflective designs. Hence, they complement classic browsing services. We note that one can use the panels to *preselect* design operators or design elements of interest for subsequent navigation in the system. That is, only preselected adaptations will be highlighted during browsing. Such selective highlighting is important for effective navigation in a system that is affected by many adaptations.

8.3 Advanced tool support

There is also a need for tool support that goes beyond browsing and navigation. In particular, we consider the following two techniques as important for a complete notion of reflective designs:

• *Interactive tool support for the life-cycle of design elements:* This boils down to dialogues per adaptation scenario. These dialogues facilitate construction, configuration, activation, and deactivation of design elements. In terms of the provided GUI, this can be compared to user interfaces of code generators for design patterns such as in the seminal work by Budinksy *et al.* [27]. However, in our setup, the dialogues facilitate runtime system adaptations.

• *Native support for debugging reflective designs:* It is inappropriate to step through the low-level code that has been inserted by system adaptations. We need to enhance the protocol for design-level reflection with respect to debugging. For instance, we need to provide means such that low-level actions can be skipped during debugging, and that design-level views are made available at breakpoints.

We are currently integrating such concepts into the *ReflectiveDesigns* framework.

8.4 Implementation of the browser adaptation

The browser adaptation itself comes down to design elements that decorate the normal browser functionality such that they add highlighting and menu entries. These decorators perform design examination, as explained earlier, to determine the actual entries to be highlighted, and the data behind the menu entries. The following fragment is the code block for the decorator that optionally highlights an item in the list of classes that are shown by the system browser:

```
classItemConversionBlock
  ↑ [:item :foo :font |
    morph ← StringMorph contents:
     item font: font.
    (Smalltalk at: item withBlanksTrimmed
     asSymbol) rdAffected
     ifTrue: [morph emphasis: rdEmphasis].
    morph]
```

That is, the block iterates over a list of menu items that represent class names. The class is extracted from the item, and then its method `rdAffected` is invoked as to check whether this class participates in any design element. If this is the case, then the `emphasis` of the item is adapted, i.e., items are turned into bold face and they are underlined; as it was shown in Fig. 10. A number of similar design elements complete the browser adaptation.

9 Related work

9.1 System adaptability by design

The conventional approach to runtime adaptability is to make adaptation opportunities explicit in the design, as discussed in Section 3. To this end, suitable design patterns such as proxy, decorator, strategy, bridge, and others are deployed in programs, class libraries, and frameworks [7]. Prominent software platforms like WebSphere [28] readily provide explicit adaptation opportunities in this manner. A sophisticated form of explicit adaptation opportunities is exemplified by Zdun's pattern language for enabling aspect-oriented idioms in conventional object-oriented programming languages [29].

In this context, the contribution of reflective designs is to make the transition from anticipated system adaptations to runtime system adaptations. We note that we emphasised the criterion of runtime adaptability, but it is clear that this is just one of the several criteria that need to be harmonised in the class libraries and software systems. Other criteria are efficiency, robustness, and safe extensibility. This tension has been analysed by Frick *et al.* [30, 31].

9.2 Technology for system adaptations

Technology for adaptations prior to runtime, e.g., byte-code engineering [32–34], compile-time reflection [35], or static weaving of aspects with AspectJ and others [36] are clearly too early for general reflective designs. Design operators require certain bits of runtime reflection. In some cases, it is convenient to define design operators in terms of higher-level aspect-oriented expressiveness rather than lower-level reflection expressiveness. While we used *AspectS* in these cases, there are similarly expressive frameworks for Smalltalk [37, 38], and for other languages as well, e.g., Handi-Wrap [39] or PROSE [5, 40] for Java. The reflective status of design elements improves on the normal situation in adapted systems, with no or only incidental information about the performed system adaptations left.

9.3 Reflective programming

In terms of expressiveness, reflective designs rely on a reflective programming setup. In this context, the contribution of reflective designs is to provide a layer for design-level reflection. This layer complements classic reflection and other recent work on reflection, e.g.:

- Static type safety, high efficiency [41–44].
- Other high-level reflection protocols [45, 46].
- Reification of messages as with composition filters by Aksit *et al.* [47].
- Advanced integration of aspects and reflection [48, 49].

9.4 Design patterns as operators or generators

The view 'design patterns as operators' has been proposed in Zimmer's dissertation [8]. Aßmann and Ludwig have further elaborated on this view, in particular by describing such operators as static meta-programs [9, 11, 50]. Meanwhile, various mechanisms have been adopted for the specification or implementation of such meta-programs, e.g., graph rewriting on a graph model of a program, metaobject protocols as in CLOS, or advanced macro mechanisms [10–13, 50]. Alternatively, one can adopt a generative approach, where program fragments for the relevant pattern are generated. This approach underlies some code generators for design patterns [27, 51, 52]. The metaprogramming approach is more versatile because it allows one to affect the existing system as opposed to the generation of a separate, new part of the system. Reflective designs make the following contributions:

- Design operators operate at runtime.
- Design decisions are traced at runtime.
- Reflective designs are not restricted to implementations of patterns.
- Design elements expose an explicit life cycle including reconfiguration.

9.5 Language support for design patterns

In [53], Bosch *et al.* clearly articulate that the implementation of design patterns in an ordinary object-oriented programming language results in problems like lack of traceability and reusability of the patterns. Very expressive languages or dedicated language support for design patterns improve on these problems. One approach to traceability is to document pattern instances and to automatically check invariants, as in Hedin's *et al.* work [54, 55]. Another approach to traceability is the modular implementation of each specific pattern instance, as in Hannemann & Kiczales' use of static weaving with AspectJ [56]. These authors also provide some reusable pattern implementations. The following approaches all facilitate some forms of reusable pattern implementations on the basis of advanced expressiveness:

- Bosch's Layered Object Model (LayOM) [57, 58].
- Ernst's statically type-safe mixins [59].
- Forbrig, Lämmel's *et al.* superimposable class structures [60, 61].
- Orleans' incremental programming with extensible decisions [62].
- Sullivan's combination of multi-dispatch, higher-orderness, and reflection [12].
- Neumann, Zdun's message filtering [63].

Our approach provides design operators on the basis of advanced language support for runtime reflection and aspect-oriented programming with dynamic weaving. As a result, implementations of design patterns are not just reusable, but they can even be deployed at runtime.

10 Concluding remarks

We have described reflective designs – an approach to impose a design-level discipline on runtime system adaptations. To this end, we integrated ideas about the implementation of design patterns, dynamic composition in aspect-oriented systems, reflection, and meta-programming. Actively running systems are adapted by the application of design operators. Each adaptation is explicitly represented by design elements. This explicit status enables a proper life cycle for system adaptations. That is, design elements can be deactivated, reactivated, and they can retire. Design elements even provide protocols for the reconfiguration, and for the examination of the performed adaptations. Reflective designs support run-system adaptations at a higher level of abstraction, when compared to techniques like basic reflection. Reflective designs support unanticipated software evolution by injecting design patterns solutions dynamically – as opposed to the conventional approach, where design patterns are implemented in a system to anticipate variation points at development time. We have implemented reflective designs in a class library for Squeak/Smalltalk. This library demonstrates the feasibility and the benefits of our approach.

We believe that *ReflectiveDesigns* and our prototypical implementation of this approach provide useful input for further research on runtime system adaptation. Also, *ReflectiveDesigns* provide a worked-out instance of the emerging trend to complement basic code browsing facilities by design-level views on software systems.

Major directions for future work are the following. Firstly, the fusion of *ReflectiveDesigns* and refactoring transformations should be completed. We note that we have focused on additive and subtractive adaptations in our work so far. Secondly, the robustness of *ReflectiveDesigns* should be improved by dedicated system analyses and rollback mechanisms. More generally, our practical approach to reflective designs needs to be complemented by formal support. Thirdly, the distance between static and dynamic design ingredients should be further decreased. At this moment, we still very much separate development-level design vs. the application of design operators for system adaptations.

11 Acknowledgment

We would like to thank the anonymous referees of the IEE Proceedings – Software; Special issue on "Reusable Software Libraries" for their substantial input on improving the structure of the paper. Thanks are due also to Uwe Zdun for comments on an earlier version of the paper.

12 References

1 Pinto, M., Fuentes, L., Fayad, M., and Troya, J.: 'Separation of coordination in a dynamic aspect oriented framework'. Proc. 1st Int. Conf. on Aspect-Oriented Software Development (AOSD), Twente, The Netherlands, April 2002, pp. 134–140
2 Akkawi, F., Bader, A., and Elrad, T.: 'Dynamic weaving for building reconfigurable software systems'. Proc. OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems, 2001
3 Hirschfeld, R., and Kawamura, K.: 'Dynamic services adaption'. Proc. 24th Int. Conf. on Distributed Computing Systems Workshops - W2: DARES (ICDCSW), Hachioji, Tokyo, Japan, 23–24 March 2004, pp. 290–297
4 Hirschfeld, R., Kawamura, K., and Berndt, H.: 'Dynamic service adaptation of runtime system extensions'. *Lect. Notes Comput. Sci.*, 2004, **2928**, pp. 227–240
5 Popovici, A., Gross, T., and Alonso, G.: 'Dynamic weaving for aspect oriented programming'. Proc. 1st Int. Conf. on Aspect-oriented Software Development (AOSD), Twente, The Netherlands, April 2002, pp. 141–147
6 Kiczales, G., des Rivieres, J., and Bobrow, D.: 'The art of the metaobject protocol' (MIT Press, Cambridge, MA, USA, 1991)
7 Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: 'Design patterns: elements of reusable object-oriented software' (Addison-Wesley, 1994)
8 Zimmer, W.: 'Frameworks und Entwurfsmuster'. PhD thesis, Universität Karlsruhe, 1997
9 Aßmann, U.: 'AOP with design patterns as meta-programming operators'. Technical Report 28, Universität Karlsruhe, Oct. 1997
10 Krishnamurthi, S., Erlich, Y.-D., and Felleisen, M.: 'Expressing structural properties as language constructs', *Lect. Notes Comput. Sci.*, 1999, **1576**, pp. 258–272
11 Ludwig, A.: 'Automatische transformation groβer softwaresysteme'. PhD thesis, Universität Karlsruhe, Dec. 2002
12 Sullivan, G.: 'Advanced programming language features for executable design patterns–better patterns through reflection'. Technical Report AIM-2002-005, MIT Artificial Intelligence Laboratory, 22 March 2002
13 von Dincklage, D.: 'Making patterns explicit with metaprogramming', *Lect. Notes Comput. Sci.*, 2003
14 Brant, J., Foote, B., Johnson, R., and Roberts, D.: 'Wrappers to the rescue', *Lect. Notes Comput. Sci.*, 1998, **1445**, pp. 396–417
15 Hirschfeld, R.: 'AspectS – aspect-oriented programming with Squeak', *Lect. Notes Comput. Sci.*, 2003, **2591**, pp. 216–232
16 Opdyke, W.: 'Refactoring object-oriented frameworks'. PhD thesis, University of Illinois, Urbana-Champaign, 1992
17 Fowler, M.: 'Refactoring: improving the design of existing code' (Addison Wesley, 1999)
18 Oreizy, P., Medvidovic, N. and Taylor, R.N.: 'Architecture-based runtime software evolution'. Proc. Int. Conf. on Software Engineering, IEEE Computer Society Press/ACM Press, 1998, pp. 177–186
19 JDrums, 'Java Distributed Run-time Updating Management System', 2003, http://www.ida.liu.se/~jengu/jdrums/
20 Evans, H., and Dickman, P.: 'DRASTIC: a run-time architecture for evolving, distributed, persistent systems', *Lect. Notes Comput. Sci.*, 1997, **1241**, pp. 275–243
21 Evans, H., and Dickman, P.: 'Zones, contracts and absorbing change: an approach to software evolution'. Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Denver, Colorado, Oct. 1999, SIGPLAN Not. 34, pp. 415–434
22 Lämmel, R.: 'A semantical approach to method-call interception'. Proc. 1st Int. Conf. on Aspect-Oriented Software Development (AOSD), Twente, The Netherlands, ACM Press, Apr. 2002, pp. 41–55
23 Lämmel, R., and Stenzel, S.: 'Semantics-directed implementation of method-call interception', *IEE Proc., Softw.*, 2004, **151**, (2), pp. 109–127
24 Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: 'Aspect-oriented programming', *Lect. Notes Comput. Sci.*, June 1997, **1241**, pp. 220–242
25 Elrad, T., Filman, R.E., and Bader, A.: 'Aspect-oriented programming: introduction', *Commun. ACM*, 2001, **44**, (10), pp. 29–32
26 Palsberg, J., and Jay, C.: 'The essence of the visitor pattern'. Proc. 22nd IEEE Int. Computer Software and Applications Conf. (COMPSAC), 19–21 Aug. 1998, pp. 9–15
27 Budinsky, F., Finnie, M., Vlissides, J., and Yu, P.: 'Automatic code generation from design patterns', *IBM Syst. J.*, **35**, (2), 1996, pp. 151–171
28 IBM, 'IBM WebSphere software platform', 2004, Web portal; http://www-306.ibm. com/software/info1/websphere/index.jsp
29 Zdun, U.: 'Pattern language for the design of aspect languages and aspect composition frameworks'. *IEE Proc., Softw.*, 2004 **151**, (2), pp. 67–83
30 Frick, A., Neumann, W., and Zimmermann, W.: 'Generation of robust class hierarchies'. Proc. Technology of Object-Oriented Languages and Systems (TOOLS) Conf., 1997, pp. 282–291
31 Frick, A., Goos, G., Neumann, R., and Zimmermann, W.: 'Construction of robust class hierarchies', *Softw.–Pract. Exp.*, 2000, **30**, pp. 481–543,
32 Austermann, M.: 'JMangler Homgepage', 2002. http://javalab.cs.uni-bonn.de/research/jmangler/index.html
33 Keller, R, and Hölzle, U.: 'Binary component adaptation', *Lect. Notes Comput. Sci.*, 1998, **1445**, pp. 307–329
34 Chiba, S.: 'Load-time structural reflection in java', *Lect. Notes Comput. Sci.*, 2000, **1850**, pp. 313–336
35 Tatsubori, M., Chiba, S., Killijian, M.-O., and Itano, K.: 'OpenJava: a class-based macro system for Java', *Lect. Notes Comput. Sci.*, 2000, **1826**
36 Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G.: 'An overview of AspectJ'. Proc. ECOOP, 2001, pp. 327–353
37 Pryor, J., and Bastán, N.: 'A reflective architecture for the support of aspect-oriented programming in smalltalk', *Lect. Notes Comput. Sci.*, 1999, **1743**
38 Böllert, K.: 'On weaving aspects', Proc. Int. Workshop on Aspect-Oriented Programming at ECOOP, 1999, pp. 301–302
39 Baker, J., and Hsieh, W.: 'Runtime aspect weaving through metaprogramming'. Proc. 1st Int. Conf. on Aspect-Oriented Software Development (AOSD), Twente, The Netherlands, April 2002, pp. 86–95
40 Popovici, A., Alonso, G., and Gross, T.: 'Just-in-time aspects: efficient dynamic weaving for java'. Proc. 2nd Int. Conf. on Aspect-oriented Software Development, 2003, pp. 100–109

41 Brandt, S., and Schmidt, R.W.: 'The design of a meta-level architecture for the BETA language'. Proc. META: presented at Workshop on Advances in Metaobject Protocols and Reflection ECOOP, Aug. 1995

42 Kleinoeder, J., and Golm, M.: 'MetaJava: An efficient run-time meta architecture for Java', in Cabrera, L.-F. and Islam, N. (Eds.): Proc. Int. Workshop on Object-Orientation in Operating Systems, 1996, pp. 54–61

43 de Oliveira Guimarães, J.: 'Reflection for statically typed languages', *Lect. Notes Comput. Sci.*, 1998, **1445**, pp. 440–461

44 Golm, M., and Kleinöder, J.: 'Jumping to the meta level: behavioral reflection can be fast and flexible', *Lect. Notes Comput. Sci.*, 1999, **1616**, pp. 22–39

45 Welch, I., and Stroud, R.: 'From Dalang to Kava – the evolution of a reflective Java extension', *Lect. Notes Comput. Sci.*, 1999, **1616**, pp. 2–21

46 Lorenz, D., and Vlissides, J.: 'Pluggable reflection: decoupling meta-interface and implementation'. Proc. Int. Conf. on Software Engineering (ICSE), 1–10 May 2003, pp. 3–13

47 Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A.: 'Abstracting object interactions using composition filters', *Lect. Notes Comput. Sci.*, **791**, 1994, pp. 152–184

48 Kojarski, S., Lieberherr, K., Lorenz, D., and Hirschfeld, R.: 'Aspectual reflection'. Workshop on Software-engineering Properties of Languages for Aspect Technologies (AOSD), 2003

49 Skotiniotis, T., Lieberherr, K., and Lorenz, D.: 'Aspect instances and their interactions'. Workshop on Software-engineering Properties of Languages for Aspect Technologies (AOSD), 2003

50 Aßmann, U., and Ludwig, A.: 'Aspect weaving by graph rewriting', *Lect. Notes Comput. Sci.*, 1999, **1799**, pp. 24–36

51 Florijn, G., Meijers, M., and Winsen, P.: 'Tool support for object-oriented patterns'. *Lect. Notes Comput. Sci.*, 1997, **1241**, pp. 472–495

52 Eden, A., Yehudai, A., and Gil, J.: 'Precise specification and automatic application of design patterns'. Proc. Int. Conf. on Automated Software Engineering, 1997, pp. 143–152

53 Bosch, J., Hedin, G., and Koskomies, K. (Eds.): Proc. LSDF– Workshop on Language Support for Design Patterns and Object-Oriented Frameworks, Research Report 6/97, University of Karlskrona/Ronneby, 1997

54 Hedin, G.: 'Language support for design patterns using attribute extension'. Bosch, J., Hedin, G., and Koskomies, K. (Eds.): Research Report 6/97, University of Karlskrona/Ronneby

55 Cornils, A., and Hedin, G.: 'Statically checked documentation with design patterns'. Proc. Technology of Object-Oriented Languages and Systems (TOOLS 33), 2000, pp. 419–430

56 Hannemann, J., and Kiczales, G.: 'Design pattern implementation in Java and AspectJ', **37**, (11); *ACM SIGPLAN Not.*, New York, ACM Press, 4–8 Nov. 2002, pp. 161–173

57 Bosch, J.: 'Design patterns & frameworks: on the issue of language support', in Bosch, J., Hedin, G., and Koskomies, K. (Eds.): Research Report 6/97, University of Karlskrona/Ronneby

58 Bosch, J.: 'Design patterns as language constructs', *J. Object-Oriented Program.*, 1998, **10**

59 Ernst, E.: 'Propagating class and method combination', *Lect. Notes Comput. Sci.*, 1999, **1628**, pp. 67–91

60 Forbrig, P., and Lämmel, R.: 'Programming with patterns'. *Proc. TOOLS-USA 2000*, 2000

61 S. Bünnig, Forbrig, P., Lämmel, R., and Seemann, N.: 'A programming language for design patterns'. Proc. GI-Jahrestagung 1999, Informatik, Reihe Informatik aktuell, 1999

62 Orleans, D.: 'Incremental programming with extensible decisions'. Proc. 1st Int. Conf. on Aspect-oriented Software Development, ACM Press, 2002, pp. 56–64

63 Neumann, G., and Zdun, U.: 'Filters as a language support for design patterns in object-oriented scripting languages'. Proc. COOTS, 5th Conf. on Object-Oriented Technologies and Systems, San Diego, California, USA, May 1999