Filters as a Language Support for Design Patterns in Object-Oriented Scripting Languages*

Gustaf Neumann and Uwe Zdun
Information Systems and Software Techniques
University of Essen, Germany
{gustaf.neumann,uwe.zdun}@uni-essen.de

Abstract

Scripting languages are designed for glueing software components together. Such languages provide features like dynamic extensibility and dynamic typing with automatic conversion that make them well suited for rapid application development. Although these features entail runtime penalties, modern CPUs are fast enough to execute even large applications in scripting languages efficiently.

Large applications typically entail complex program structures. Object-orientation offers the means to solve some of the problems caused by this complexity, but focuses only on entities up to the size of a single class. The object-oriented design community proposes design patterns as a solution for complex interactions that are poorly supported by current object-oriented programming languages. In order to use patterns in an application, their implementation has to be scattered over several classes. This fact makes patterns hard to locate in the actual code and complicates their maintenance in an application.

This paper presents a general approach to combine the ideas of scripting and object-orientation in a way that preserves the benefits of both of them. It describes the object-oriented scripting language XOTcl (*Extended* OTcl), which is equipped with several language functionalities that help in the implementation of design patterns. We introduce the filter approach which provides a novel, intuitive, and powerful language support for the instantiation of large program structures like design patterns.

*Published in: "*Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*", San Diego, May 3-9 1999.

1 Introduction

1.1 Scripting Languages

In applications, where the emphasis lays on the flexible reuse of components, scripting languages, like Tcl (Tool Command Language [25]), are very useful for a fast and high-quality development of software. The application development in scripting languages differs fundamentally from the development in systems programming languages [26] (like C, C++ or Java), where the whole system is developed in a single language. A scripting language follows a two-level approach, distinguishing between components (reusable software modules) and glueing code, which is used to combine the components according to the application needs. This two level approach leads to a rapid application development [26].

Scripting languages are typically interpreted and use a dynamic type system with automatic conversion. The application developer uses a single data type (strings) for the representation of all data. Therefore, the interfaces of all components fit together automatically and the components can be reused in unpredicted situations without change. The disadvantages of scripting languages are a loss in efficiency (e.g. for dynamic conversions and method lookup) and the lack of reliability properties of a static type system [18]. But these disadvantages can be compensated to a certain degree:

- For several application tasks, the loss of efficiency is not necessarily relevant, because the time critical code can be placed into components written in efficient systems programming languages. Only the code to control these components is kept in the highly flexible scripting language.

- Since the components are typically written in a language with a static type system the reliability argument applies only on the glue code, used to combine the components. To address these remaining problems we have integrated an assertion concept based on pre- and post-conditions and invariants (see [24]).

Since TCL is designed for glueing components together, it is equipped with appropriate functionalities, such as dynamic typing, dynamic extensibility and read/write introspection. Many object-oriented TCL-extensions do not support well these abilities in their language constructs. They integrate foreign concepts and syntactic elements (mostly adopted from C++) into TCL (see e.g. [15, 9]). Even less appropriate is the encouraged programming style in structured blocks and the corresponding rigid class concept, which sees classes as write-once, unchangeable templates for their instances. This concept is not compatible with the highly dynamic properties of the underlying scripting language.

1.2 Scripting and Object Orientation

The three most important benefits of object-orientation are encapsulation of data and operations, code reuse through inheritance, and polymorphism. These should help to reduce development time, to increase software reuse, to ease the maintenance of software and to solve many other problems.

But these claims are not undoubted: For example Hatton [11] argues that the non-locality problems of inheritance and polymorphism in languages, like C++, do not match the model of the human mind well.

Encapsulation lets us think about an object in isolation; this is related to the notion of manipulating something in short-term memory exclusively. Therefore, encapsulation fits the human reasoning. Since in scripting languages a form of code reuse is already provided through reusable components, the foremost reason for the use of object-orientation in a scripting language is the encapsulation. For that reason, the inheritance problem also seems less conflicting, because the inheritance is mainly used to structure the system and to put the components together properly. Inheritance in scripting applications normally does not lead to large and complex classes that are strongly dependent on each other.

Hatton [11] criticizes the polymorphism in C++ as damaging, because objects become more difficult to manipulate through the evolving non-locality in the structures. They involve a pattern-like matching of similar behavior in long-term memory. The string as an uniform and flexible interface instead of the use of polymorphism makes the objects easier to be put together. They get one unique behavior *and* one unique interface. They may be used in different situations differently, but the required knowledge about the object remains the same.

These arguments account for the glueing idea of the scripting language in the scope of a single class and its environment. This scope of a "programming in the small" is the strength of current object-oriented (language) concepts. Their weakness is the "programming in the large", where all components of a system have to be configured properly. The concepts only provide a small set of functionalities that work on structures larger than single classes, e.g. from languages like Java or C++ the following are known:

- virtual properties are used to define additional object- and class-properties,
- abstract classes specify formal interfaces and requirements for a set of classes,
- parametric class definitions are used for different data types on one class-layout.

Beneath such language constructs, methodical approaches, like frameworks, exist. Since they are coded using conventional language constructs the problems due to the language insufficiencies are not eliminated. The main insufficiency is that classes and objects are relatively small system-parts compared to an entire, complex system. Therefore, the wish for a language construct, which maps such a large structure to an instantiable entity of the programming language, arises.

1.3 OTcl – MIT Object Tcl

We believe OTCL [32] is an extraordinary object-oriented scripting language which supports several features for handling complexity. It preserves and extends the properties of TCL like introspection and dynamic extensibility. Therefore, we used OTCL as the starting point for the development of XOTCL.

In OTCL each object is associated with a class. Classes are ordered by the superclass relationship in a directed acyclic graph. The root of the class

hierarchy is the class `Object` that contains the methods available in all instances. A single object can be instantiated directly from this class. In OTcl classes are special objects with the purpose of creating and managing other objects. Classes can be created and destroyed dynamically like regular objects. Classes contain a repository of instance methods ("instprocs") for the associated objects and provide a superclass relationship that supports multiple inheritance.

Since a class is a special (managing) kind of object, it is managed by a special class called "meta-class" (which manages itself). The meta-class is used to create and to instantiate ordinary classes. By modifying meta-classes it is possible to change the behavior of the derived classes widely. All inter-object and inter-class relationships are completely dynamic and can be changed at arbitrary times with immediate effect.

2 Design Patterns in Scripting Languages

The complexity of many large applications is caused by the combination of numerous, often independently developed components, which have to work in concert. Typically, many classes are involved, with different kinds of non-trivial relationships, like inheritance, associations, and aggregations. Design patterns provide abstractions over reusable designs, that can typically be found in the "hot spots" [28] of software architectures. Patterns are designed to manage complexity by merging interdependent structures into one (abstract) design entity.

Design patters are considered increasingly often as reusable solutions for general problems. Specialized instances of design patterns can be used in a diversity of applications. Soukup [30] defines a pattern as follows:

> *"A pattern describes a situation in which several classes cooperate on a certain task and form a specific organization and communication pattern."*

Design patterns are collected in pattern catalogs [10, 6]. Typically, these catalogs contain general patterns, but there are also catalogs which collect domain specific patterns. In this paper, we see a design pattern as an abstract entity with normative, constructive and descriptive properties, that is identified in the design process and has to be preserved (with documentation and usage constraints) in the implementation as well.

2.1 Language Support for Design Patterns

Most efforts in the literature of design patterns collect and catalog patterns. These activities are very important, since they are the basis for new software architectures using design patterns. Soukup [30] remarks that this basic work is not yet ended.

Most authors present design patterns as guidelines for the design. When they are used in the design phase, the abstract pattern has to be transformed into a concrete implementation for each usage. A basic idea of this paper is to allow one to code a pattern once in an abstract way (e.g. for a pattern-library) and reuse it later in a specialized manner. The gained advantage is that patterns become (abstract) entities of the design process as well as of the implementation. This is similar to the use of the design process entities "object" and "class" which are also entities in object-oriented programming languages.

There are only a few efforts in the direction of language support for design patterns so far. We believe that one reason for this lack of support is due to the targeted languages. Conventional object-oriented programming languages, like C++, offer no support for reproduction of larger structures than classes (like design patterns). Therefore, it is nearly impossible to get a sufficient reproduction of such structures as an entity[1]. But there are more reasons [4], why language support for design patterns should be improved:

- *Traceability:* The pattern is scattered over the objects and, therefore, hard to locate and to trace in an implementation.
- *Self-Problem:* The implementation of several patterns requires forwarding of messages, e.g. an object A receives a message and forwards it to an object B. Once the message is forwarded, references to *self* refer to the delegated object B, rather than to the original receiver A. (known as the *self-problem* [17]).
- *Reusability:* The implementation of the pattern must be recoded for every use.

[1] Soukup [30] shows that some design patterns can be implemented as classes in C++ using **friend**, but Bosch [4] points out that these are only a few.

- *Implementation Overhead:* The pattern implementation requires several methods with only trivial behavior, e.g. methods solely defined for message forwarding.

Pree [28] identifies seven meta-patterns that define most of the patterns of Gamma et.al. [10]. This indicates that it is possible to find language constructs, which are able to represent all structures definable by these meta-patterns. In this work, we present the *filter* as such a language construct.

2.2 Language Support for Design Patterns in XOTcl

In the following sections, we describe the language support for design patterns we have developed. We introduce our ideas with examples from the language *Extended* OTcl (XOTcl, pronounced *exotickle*) which is an extension of OTcl, but we give no introduction to the language. Figure 1 shows the relationship between XOTcl and OTcl and lists important properties of OTcl.

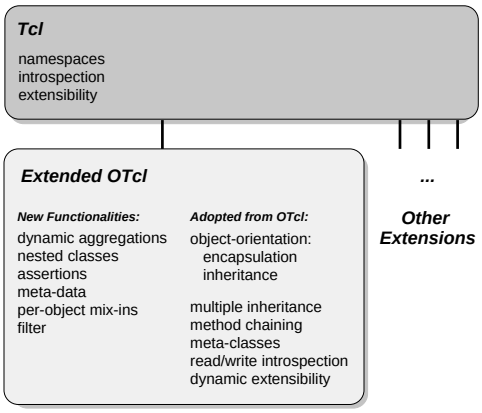

Figure 1: Language Extensions of XOTcl

Tcl and OTcl already have many properties that are very helpful for the implementation of patterns. Dynamic typing, as stated above, eases the management of highly generic structures. The definition of pattern parts as meta-classes makes them entities of the programming language and instantiable with the name of the pattern. Introspection allows self-awareness and adaptive programs, and simplifies the maintenance of relationships such as aggregations. Per-object specialization eases implementation of single objects with varying behavior, e.g. non-specializable singleton patterns.

In addition to the abilities of OTcl, we implemented in XOTcl new functionality specially targeted on complex software architectures and patterns. In particular, we added:

- *nested objects* based on Tcl's namespaces to (a) reduce the interference of independently developed program structures, (b) to support nested classes and (c) to provide dynamic aggregations of objects.
- *assertions* to reduce interface problems, to improve the reliability weakened by dynamic typing and, therefore, to ease the combination of many components,
- *meta-data* to provide self-documentation of objects and classes,
- *per-object mixins* as a flexible means to give an object access to several different addition-classes, which may be changed dynamically, and finally,
- *filters* as a means of abstractions over method invocations to implement patterns (see Section 2.3).

The first three extensions are variations of known concepts, which we have adopted in a dynamical and introspective fashion, the last two are both novel approaches. In [24, 33] we describe all these features in detail, in this paper we solely describe the filter approach.

2.3 The Filter Approach

We have pointed out that the realization of design patterns as entities is a valuable goal and that the object-oriented paradigm is not able to achieve this through classes alone. OTcl offers a means for the instantiation of large structures, like entire design patterns: the meta-classes. But in pure OTcl only a few patterns are instantiable this way (e.g. the abstract factory as in [33]), without suffering from the problems stated in Section 2.1. Typically, these patterns do not rely on a delegation or aggregation relationship.

Even though object-orientation orders program structures around the data, objects are characterized primarily by their behavior. Object-oriented programming style encourages the access of encapsulated data only through the methods of the object, since this allows data abstractions [31]. A method invocation can be interpreted as a message exchange between the calling and the called object. Therefore,

objects are only traceable at runtime through their message exchanges. At this point the filters can be applied, which are able to catch and manipulate all incoming and outgoing messages of an object.

> *A filter is a special instance method registered for a class C. Every time an object of class C receives a message, the filter is invoked automatically.*

A filter is implemented as an ordinary instance method (`instproc`) registered on a class. When the filter is registered, all messages to objects of this class must go through the filter, before they reach their destination object. The filter is free in what it does with the message, especially it can (a) pass (the potentially modified) message to other filters and finally to the object, or (b) it can redirect it to another destination, or (c) it can decide to handle the message solely.

The forward passing of messages is implemented as an extension of the `next` primitive of OTcl. `next` implements method chaining without explicit naming of the "mixin"-method. It mixes the same-named superclass methods into the current method (modeled after CLOS [5]). All classes are ordered in a linear next-path. At the point marked by a call to `next` the next shadowed method on this next-path is searched and, when it is found, it is mixed into the execution of the current method.

In XOTcl, a single class may have more than one filter. All the filters registered for a class form an ordered filter chain. Since every filter shadows all instance methods, `next` appears as a suitable mechanism to call the next filter in the chain. When all filters are worked through, the actual called method is invoked. By placement of the `next`-call, a filter defines if and at which point the remaining filters (and finally the actual method-chain) are invoked.

```
Class A
A instproc Filter-1 args {
  puts "pre-part of [self proc]"  ;# pre part
  next                            ;# next call
  puts "post-part of [self proc]" ;# post part
}
A filter Filter-1
A a1
a1 set x 1
```

This introductory example defines a single class and a single filter instproc. It registers the filter for the class using the `filter` instance method. An object `a1` is created. In the last line the predefined `set` method is invoked. Automatically the registered filter `Filter-1` of class `A` receives the message `set`. The filter instproc consists of three (optional) parts: The *pre-part* consists of the actions before the actual method is called, the `next` call invokes the message chaining, and the *post-part* contains the actions to be executed before the filter is left. In this example the pre- and post-parts are simple printing statements, but in general they may be filled with arbitrary XOTcl-statements. The distinction between the three parts is just a naming convention for explanation purposes.

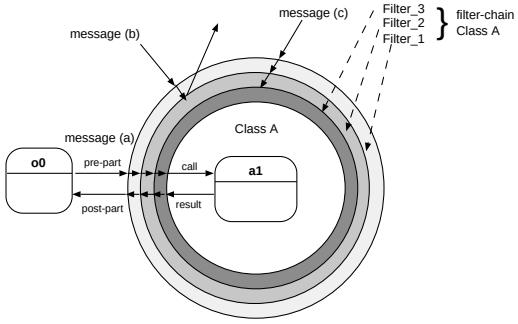

Figure 2: Cascaded Message Filtering

The following extension of the introductory example shows how to apply more than one filter, which are cascaded through `next` (see Figure 2). In this extended example a filter chain consisting of two filters is used. Again `next` forwards messages to the remaining filters in the chain or to the actual called method. The method `filter` registers the list of filters to be used.

```
A instproc Filter-2 args {
  puts "only a pre-part in [self proc]"
  next
}
A instproc Filter-3 args {
  next
  puts "only a post-part in [self proc]"
}
A filter {Filter-1 Filter-2 Filter-3}
```

When an instance `a1` of class `A` receives a message, like "`a1 set x 1`", it produces the following output. The `next`-call in the last filter `Filter-3` of the chain invokes the actual called method `set`.

```
pre-part of Filter-1
only a pre-part in Filter-2
only a post-part in Filter-3
post-part of Filter-1
```

The `filter` method can be used to remove filters dynamically as well. E.g. the filters `Filter-1` and `Filter-3` can be removed by:

```
A filter Filter-2
```

On each class the filters are invoked in the order specified by the `filter` instance method. To avoid circularities all filters which are currently active – that means that the current call is invoked directly

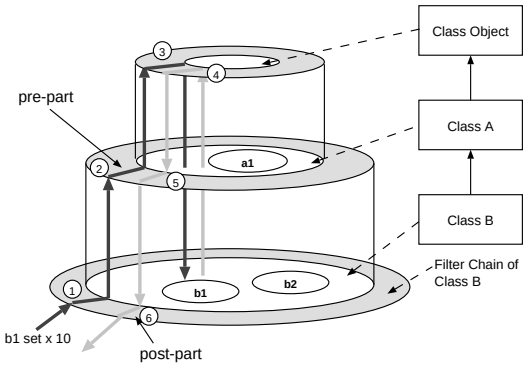

Figure 3: Filter Inheritance

or indirectly from a filter instproc – are temporarily left out of the filter chain. Filter chains can also be combined through (multiple) inheritance using `next`. Since filters are normal instprocs they may themselves be specialized through inheritance. When the end of the filter chain of the object's class is reached, the filter chains of the super-classes are invoked using the same precedence order as for inheritance.

This is demonstrated by the example displayed in Figure 3. `B` is a subclass of `A` with two instances `b1` and `b2`. Both instances are filtered with the chains registered on `B`, `A` and `Object`. The invocation `b1 set x 10` results in the next-path shown in Figure 3.

```
Class B -superclass A
B instproc Filter-B args {
  puts "entering method: [self proc]"
  next
}
B b1; B b2
B filter Filter-B
b1 set x 10
```

Filters have rich introspection mechanisms. Each class may be queried (using the introspection method `info filters`) what filters are currently installed. A filter method can obtain information about itself and its environment, and also about the calling and the called method. Examples are the name of the calling and the called method, the class where the filter is registered, etc. (see for details [24]). By using these introspection mechanisms filters can exploit various criteria in order to decide how to handle a message.

Often it is useful to add filters to an existing chain of filters. This can be achieved conveniently by the instproc `filterappend` defined for the top-level class `Object`. Therefore this method is inherited by all classes.

```
Object instproc filterappend f {
  [self] filter [concat [[self] info filters] $f]
}
A filterappend {Filter-2 Filter-3}
```

3 Language Support for Design Patterns using Filters

Now we present a systematic approach how filters can be used to implement design patterns. In general, filters are very flexible and well-suited for implementing patterns in various creative approaches.

3.1 Applying Filters on Meta-Patterns

In [28] Pree has identified meta-patterns as structures underlying several design pattern. They subdivide the pattern into a general, generic pattern-class (called *template*) and a class which serves as an anchor for the application specific details (called *hook*).

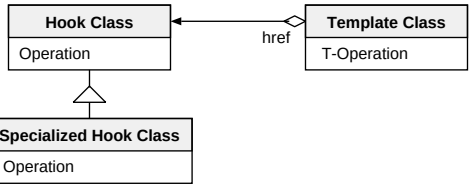

Figure 4: The 1:1 Meta-Pattern [28] with a Specialized Hook

Figure 4 shows a simple meta-pattern, that is based on a 1:1 association. It separates the specializations of a hook class from a template class. This is just an example pattern to give an idea of hook and template. It is obvious that many object-oriented structures, like several design patterns in [10, 6], are based upon this meta-structure.

The methods of the template implement the generic part of the structure and invoke the hook methods. The abstract hook forms a common interface for its specializations. The structure can be reused with different special hooks without changes to the template.

Filters are well-suited to implement meta-patterns. By using a filter all activities of a pattern can be treated in one entity (the filter instance method). Since all messages are directed to the filter the abstract tasks of the pattern can be separated from actual tasks of the application. But this alone would not be a reusable solution, since for every template class of every task, where the pattern could be used, a new filter method would have to be implemented. In order to achieve reusability we use a meta-class that provides the desired functionality. This meta-class may be stored in a library and can be reused every time a similar problem occurs.

The steps, to obtain a reusable and instantiable pattern based on filters from a pattern class diagram (e.g. Figure 4), are:

1. Find the hook and template classes.
2. Create a meta-class under the general name of the pattern.
3. Add a filter method to the meta-class, which performs all recurring tasks desired from the design pattern (especially the tasks of the template).
4. Add a constructor to the meta-class, which registers the filter on classes derived from the meta-class (and performs pattern specific initialization tasks).
5. Add additional methods to the meta-class (e.g. like registration of special hooks) to avoid hard-coding of pattern semantics in the filter method.

With slight adaptations this scheme is applicable on all patterns that rely on Pree's meta-patterns (e.g. most of the patterns in [10]). But nevertheless most other patterns, since they normally involve messages exchanges, are supportable by filters. A meta-class can be defined as a general solution for a large number of related problems. In order to use it, the application must derive a class from it (e.g. with the name of the template class) and concretize the application specific actions (that means the hook classes).

Now we show on a template for the 1:1 meta-pattern, how to apply the scheme in XOTcl. The first step is to define a meta-class. In XOTcl a meta-class is defined by referencing the meta-class `Class` as superclass of a newly defined class:

```
Class 1-1-Meta-Pattern -superclass Class
```

Secondly, a filter instproc must be defined:

```
1-1-Meta-Pattern instproc 1-1-Filter args {
  # filters actions
  # e.g. forwarding messages to the special hook
}
```

As the next step the constructor `init` registers the filter on the newly created class and performs other initialization tasks, like variable initialization, method declaration, etc.:

```
1-1-Meta-Pattern instproc init args {
  # initialization tasks
  [self] filterappend 1-1-Filter
}
```

For real applications the meta-class has to be extended with additional methods. In order to complete the implementation of the 1:1 meta-pattern a method, which stores a reference to the special hook on the object, has to be defined. Finally, the meta-pattern is instantiated to create a filtered template class.

```
1-1-Meta-Pattern FilteredTemplate
```

In order to provide the hook for the filter a special hook class and perhaps concretizations have to be created.

The presented scheme may be extended for more specialized patterns, e.g. a recursive pattern may require recursive registering of the filter. Sometimes it is useful (but not necessary) to apply a second filter (e.g. in patterns with a second referencing relationship, like mediator or observer in [10]).

3.2 Design Pattern Examples

The idea underlying meta-patterns splits patterns into two parts: the template and the hook. We have shown a scheme how to apply a filter if this division is possible. This section applies the scheme in order to implement three example patterns from [10].

3.2.1 The Adapter Pattern

The adapter pattern [10] converts the interface of a class into another interface that a client expects. Therefore, an adapter is a means to let classes cooperate despite of incompatible interfaces. As shown in Figure 5 the conventional solution is to forward the messages from `Adapter` to `Adaptee` by explicit calls. This approach entails that for every adapted method a new additional method must be defined in the adapter. This leads to an implementation overhead. Moreover, the solution's program code is neither reusable nor traceable.

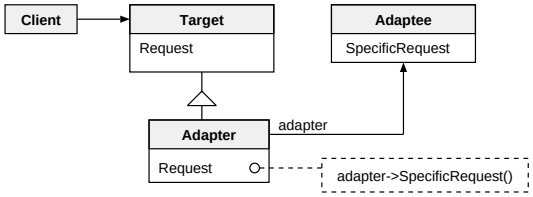

Figure 5: The Adapter Pattern [10]

The solution for the adapter problem presented below is based on filters and avoids these problems.

It is reusable and does not require the implementation overhead resulting from methods which are defined solely for the purpose of message forwarding. The forwarding is handled automatically by the next primitive in the filter method, no additional helper methods are needed.

By following the systematic steps presented above, we identify the template (here `Adapter`) and the hook (here `Adaptee`). The adapter pattern resembles the 1:1 meta-pattern of Section 3.1, but it has no special hooks. The desired actions of the template are to forward requests to specific requests. This will be handled by the filter. Firstly, we define a meta-class which replaces the pattern from the conventional design in Figure 5.

```
Class Adapter -superclass Class
```

A meta-class can be used to derive new classes that can access the instance methods of the meta-class. The derived classes are constructed with constructor of the meta-class. Next, we define the filter instance method:

```
Adapter instproc adapterFilter args {
  set r [[self] info calledproc]
  [self] instvar specificRequest adaptee \
    [list specificRequest($r) sr]
  if {[info exists sr]} {
    return [eval $adaptee $sr $args]
  }
  next
}
```

The `info calledproc` command returns the originally called method. This is the general request which is to be mapped to a specific request. The two variables `specificRequest` and `adaptee` are instance variables which are linked to the current scope by the primitive method `instvar`. The `specificRequest` for the called method is mapped to the variable `sr`. `adaptee` is the object which handles the specific requests. If there exists a mapping of the current request, the filter forwards the message to the associated method. Otherwise the message is not redirected, but passed further on by the filter along the next-path.

As the next step we have to define the constructor which adds the filter to the class. In order to be able to set the `specificRequest` and `adaptee` variables it is convenient to define instprocs for this purpose, which are defined for the derived classes. These instprocs are created dynamically by the constructor (the `init` instproc) of the meta-class:

```
Adapter instproc init args {
  [self] filterappend [self class]::adapterFilter
  next
  [self] instproc setRequest {r sr} {
    [self] set specificRequest($r) $sr
  }
  [self] instproc setAdaptee {a} {
    [self] set adaptee $a
  }
}
```

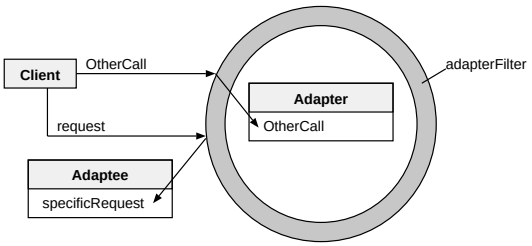

Figure 6: The Adapter Pattern Using Filters

Now the abstract pattern is converted into a meta-class, which can be used to derive classes with the behavior of the pattern: method invocations, which correspond to registered requests, are redirected to the adaptee object; all other invocations are passed unmodified to the object through the next-path (see Figure 6).

The solution in [10] suffers from the self-problem, since the originally called object of the adapter class is not the object which performs the desired task. This problem is not addressed by the filter solution presented above. A more sophisticated solution, which does not suffer from the self-problem, is to define the filter on the adaptee instead of the adapter. For the sake of simplicity we presented here the slightly simpler version.

A sample application of this pattern is a class which handles network connections. Derived classes, like `FTP`, `HTTP`, etc. allow one to handle specialized connections. All of them must implement a method `connect`. A method `discard` of the `Connection` class is able to close connections of all different kinds. Suppose a FTP connection routine from a library class with a different interface should be used. A filter adapter on basis of the defined meta-class can solve this problem elegantly. Firstly the interfaces of the related classes:

```
# interface of the library class
Class FTPLIB
FTPLIB instproc FTPLIB_connect args {...}

# the connection class
Class Connection
# an abstract connection method
Connection instproc connect args {...}
# the method to close a network connection
Connection instproc discard args {...}

... other class definitions, like HTTP
```

Now we derive a class FTP from the Adapter. The meta-class's constructor defines the two convenience

methods and registers the filter on the new class FTP automatically. Strictly speaking the convenience methods are not necessary, but they provide a simpler interface. The class FTP has a constructor that automatically creates an associated adaptee and provides the needed information for the filter though the convenience methods.

```
Adapter FTP -superclass Connection
FTP instproc init args {
  FTPLIB ftpAdaptee
  [self] setRequest connect FTPLIB_connect
  [self] setAdaptee ftpAdaptee
}
```

Finally, the FTP class can be used and is adapted automatically. Since only the method `connect` was a registered request, all `discard`-calls reach the `Connection` class.

```
FTP ftp1
ftp1 connect
...
ftp1 discard
```

This simple example can be extended with only a few more lines of code to provide more sophisticated adaptations (e.g. altering parameters, adapting to other objects, etc) without architectural redesign.

3.2.2 The Composite Pattern

A recursive pattern from [10] is the composite pattern, shown in Figure 7. The composite pattern helps to arrange objects in hierarchies with a unique interface type, called component. The objects are arranged in trees with two kinds of components: leafs and composites. Every composite can hold other components.

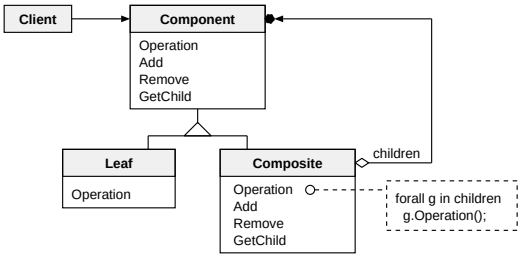

Figure 7: The Composite Pattern [10]

There are several disadvantages in the implementation of the pattern in [10]. The composite pattern structure contains dynamic object aggregation, what is not provided in C++. Therefore, the implementation lacks flexible mechanisms to handle and introspect the aggregation. An implementation overhead results from the necessity to define methods for management of the dynamic aggregation. Furthermore, the scattering of the pattern across several classes, leads to a mixing of application and pattern structures that reduces reusability.

The pattern (as presented in [10]) is not an abstract entity; therefore, it is hard to specialize and to reuse it. Also, it is not easy to find it in source code, if it is not well commented, and both description in the pattern classes and the runtime object structure are hard to introspect and not traceable.

In order to implement the pattern as a filter, we firstly identify its elements. The composite class forms the template, the component class the hook. The desired action of the template is to forward all messages to the aggregated objects recursively. The application specific actions are the concretizations that determine what these classes do with the messages. We create a meta-class:

```
Class Composite -superclass Class
Composite instproc addOperations args {...}
Composite instproc removeOperations args {...}
```

As a useful enhancement to the solution in [10], new operations are added and removed by `addOperations` and `removeOperations` (not to be confused with the methods for aggregation handling in Figure 7). Only registered operations will be forwarded to the objects in the composite patterns runtime structure.

All generic pattern tasks will be performed by a filter. It handles the forwarding to the components of a composite:

```
Composite instproc compositeFilter args {
  [[self] info class] instvar operations
  set r [[self] info calledproc]
  if {[info exists operations($r)]} {
    foreach object [[self] info children] {
      eval [self]::$object $r $args
    }
  }
  return [next]
}
```

In the composite filter firstly the request is compared to the operations in the `operations`-list. If the request is a registered operation, the message is forwarded to the child. Though children may be composites, this mechanism functions recursively on the entire structure, until the leaves are reached.

In order to register the filter on a new composite class automatically, we append it in the constructor of the meta-class:

```
Composite instproc init {args} {
  next
  [self] filterappend Composite::compositeFilter
}
```

Now we will show on an illustrative example that this single method handles *all* semantics of the pattern. As a sample application we will build up a simple graphic:

```
Class Graphic
Graphic instproc draw {} {...}
```

Different graphics objects can be defined on basis of the component type. For example we can define a Composite (Picture) with two leaves (Line and Rectangle):

```
Composite Picture -superclass Graphic
Class Line -superclass Graphic
Class Rectangle -superclass Graphic
```

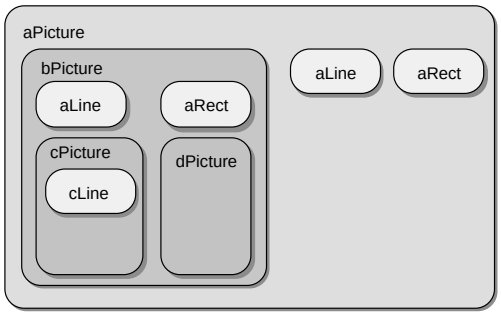

Figure 8: The Composite Object Structure

The graphic structure shown in Figure 8 can be constructed by:

```
Picture aPicture
Picture aPicture::bPicture
Line aPicture::aLine
Rectangle aPicture::aRect
Line aPicture::bPicture::aLine
Rectangle aPicture::bPicture::aRect
Picture aPicture::bPicture::cPicture
Picture aPicture::bPicture::dPicture
Line aPicture::bPicture::cPicture::cLine
```

An invocation of the `draw` method on a complex object, like:

```
Picture addOperations draw
```

registers the `draw` message for all the component objects in the structure. A call of `draw` draws the whole hierarchy:

```
aPicture draw
```

Note how simple and short it was to instantiate the sample application. Beneath the elimination of the problems mentioned above, compared to a solution of the picture application following [10], the filter solution is much shorter and easier to understand. It avoids complex structures that are connected in many ways, and removes the need for replicated code, since it takes the pattern semantics completely out of the application. Furthermore, the result is that the pattern is reusable as a program fragment (and may be put into a library of patterns) and not only as a design entity, which has to be recoded for every usage.

To map the recursive structure of the pattern a more general solution, in which each composite class gets recursively its own filter instproc, is easily achievable. This would allow one to specialize the filters behavior for certain branches of the structure (e.g. in order to fade out parts of the picture) or to store different composites, components or other classes in the pattern structure (what is possible to certain degree in the solution presented).

3.2.3 The Observer Pattern

The observer pattern presented in this section fulfills the task of informing a set of depending objects ("observers") of state changes in one or more observed objects ("subjects"). This problem is well known and often addressed, e.g. by the publisher subscriber pattern [6] or Model-View-Controller [14]. Figure 9 shows the observer design pattern as presented in [10].

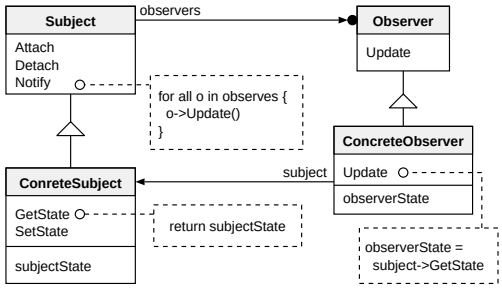

Figure 9: The Observer Pattern [10]

Bosch [3] identifies the problem that the traceability of the pattern suffers from the fact that the methods `attach`, `detach` and `notify` do not build up a conceptual entity and that the calls of `notify` must be inserted at every point where a state change occurs. The reusability of the concrete subjects also suffers from these problems. A filter, which directs all state changes of the subject to the observers does not have these problems and provides a reusable solution.

In order to implement an observer pattern based on filters we create meta-classes for the observer and the subjects. The subjects are structured as nested class to preserve the unity of the pattern:

```
Class Observer -superclass Class
Class Observer::Subject -superclass Class
```

In this example we only handle the relationship between subject (as template) and observer (as hook) by a filter. In a more sophisticated solution the

second referencing relationship between concrete observer and concrete subject may also be replaced by a filter. Now we can define a filter which handles the notification:

```
Observer::Subject instproc notifyFilter args {
  set r [[self] info calledproc]
  [self] instvar preObservers postObservers \
         [list preObservers($r)  preObs] \
         [list postObservers($r) postObs]
  if {[info exists preObs]} {
    foreach o $preObs {$o update [self] $args}
  }
  set result [next]
  if {[info exists postObs]} {
    foreach o $postObs {$o update [self] $args}
  }
  return $result
}
```

Observers are registered with the `attach` and `detach` methods. As a special feature we allow both pre- and post-observers to be registered. When the filter method is invoked, firstly all registered pre-observers are informed, then the actual method is invoked and then all post-observers are informed. Finally, the filter returns the result of the called method.

The trivial methods to register or unregister observers (here: `attachPre`, `attachPost`, `detachPre` and `detachPost`) are created by the constructor `init` on all instantiated classes, so that their objects can reach them as `instproc`'s (not presented here).

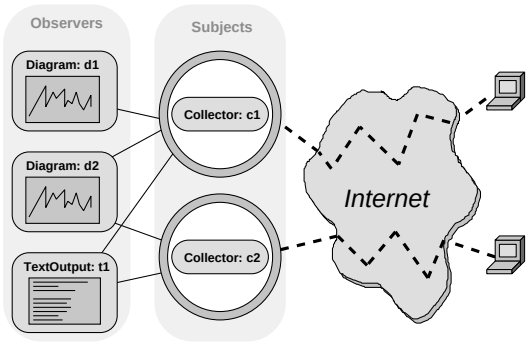

Figure 10: Observer Example

We demonstrate the usage of the abstract observer pattern by an example of a network monitor which observes a set of connections and maintains several views on these (e.g. a diagram and a textual output). In the implementation the class `Pinger` encapsulates the view and collector classes, the collectors are treated as subjects of the observer:

```
Class Pinger
Observer::Subject Pinger::Collector
Observer Pinger::Diagram
Observer Pinger::TextOutput
```

The `Collector` starts the observation of the network connection in its constructor, e.g.:

```
Pinger::Collector instproc init args {
  set hostName 132.252.180.67
  set f [open "| /bin/ping $hostName" r]
  fconfigure $f -blocking false
  fileevent $f readable "[self] ping \[gets $f\]"
}
```

The operation `ping` is the network event, which must be handled by the collector. Since the collector is a concrete subject it needs a method (`getResponse`) which is invoked by the observers to get its current state:

```
Pinger::Collector instproc ping {string} {...}
Pinger::Collector instproc getResponse {} {...}
```

The two observers must concretize their `update` methods. Both must catch the actual state of the subject using `getResponse` and then they will update their presentation. The text output presentation may look like:

```
Pinger::TextOutput instproc update {subject args} {
  set response [$subject getResponse]
  puts "PINGER: $subject --- $response"
}
```

For concrete applications the classes must be instantiated. Here are two collectors, some observers and some attachments:

```
Pinger::Collector c1
Pinger::Collector c2
Pinger::Diagram d1
Pinger::Diagram d2
Pinger::TextOutput t1

c1 attachPre ping d1 d2
c1 attachPost ping d2 t1
c2 attachPost ping t1 d2
```

This attaches the diagrams and the text output to the collectors `c1` and `c2` as pre- and as post-observers, as shown in Figure 10.

4 Related work

There are many other concepts with names containing the word "filter" (e.g. in the area of mobile/distributed computing [27, 16]). The composition filter model [1] introduces the idea of a higher-level object interaction model through abstract communication types (ACTs). Besides such basic ideas of a means to change, redirect, or otherwise affect messages, we have not found an approach with comparable properties like filters (as user-defined methods, mixin of filter chains, inheritance, etc.). Nevertheless, in Section 4.3, we describe other approaches providing language support for design patterns.

4.1 Meta-Object-Protocol

One of the most flexible environments for object-oriented engineering is the CLOS environment with its meta-object-protocol [13]. We are convinced that filters can be implemented in this environment which provides many hooks to influence the behavior and semantics of objects. Our filter approach differs significantly, since filters provide a high level construct, which is tailored to monitor and to modify object interactions.

One example in [13] enhances CLOS with encapsulated methods capable of restricting the access of private variables to methods of their class. The system's method, used to apply methods, is enhanced with a sub-protocol which can add a set of function bindings to the method body's lexical environment. The filter would have been a shorter and higher level solution for this problem, because it does not require modifications or additions to the underlying systems behavior. Therefore it does not require knowledge about the systems structure, like how the system applies methods or how lexical definitions are bound to methods. Moreover, the filter solution can easier be scaled, since filters may be dynamically registered and unregistered.

4.2 Meta-Programming

From the abstraction point of view filters are closely related to the area of meta-programming, which was studied in the area of lisp-like languages (e.g. [2]) or in the area of logic languages, as sketched in this section.

The filter approach is a very general mechanism which can be used, besides language support for design patterns, in various other application areas. We see object-orientation and filters as an analogy to the interpretation layer introduced by meta-programs which are used to interpret existing programs in a new context with additional functionality [20, 21]. In [22] the abstraction introduced by layered interpreters is called interpretational abstraction. The basic idea of interpretational abstraction is to treat program instructions of one program (source program) as data of another program (a meta-program, a compiler or interpreter) that reasons about the instructions of the source program. During this reasoning process new functionality can be introduced into the source program by interpreting the goals of the source program in a new context. Instead of altering the application program (the knowledge representation), an additional interpretation layer can be introduced to change the behavior in certain situations. This way interpreters can be used as a programming device. The inefficiency of the reasoning process can be eliminated by techniques like partial evaluation [8] or interpreter directed compilation [21].

The filter approach is an introduction of meta-programming ideas into object-orientation. Even if the filter never accesses the real program (which a filter in XOTcl could do through the provided introspection mechanisms), it has full and unlimited access to the most important thing in object-oriented runtime structures: the messages. A filter handles messages of objects as data which can be processed in arbitrary ways (modified, redirected, handled). The filters are able to reinterpret messages freely, the filter methods are "interpreters" for messages and can influence all object communication.

In general the application domain of filters is very wide. For example assertions and meta-data as presented in [24] could have been implemented using filters. The only argument against this was, that the implementation in C is much faster than implementation using filters, since in the current implementation they reduce execution speed. However, it would be interesting to investigate, to what degree compilation methods like these described above could eliminate the overhead.

4.3 Other Approaches for Supporting Design Patterns

As stated above, Soukup [30] has identified problems in the implementation of popular design patterns [10] and has shown that some patterns could be implemented as classes.

The LayOM-approach [3] is the most similar to the filter approach. It offers an explicit representation of design patterns using an extended object-oriented language. The approach is centered on message exchanges as well and puts layers around the objects which handle the incoming messages. Every layer offers an interface for the programmer to determine the behavior of the layer through a set of operators which are (statically) given by the layer definition. LayOM is a compiled language with a static class concept and can be translated into C++. The model is statically extensible with new layers.

The filter approach differs from LayOM since it can represent design patterns as normal classes and needs no new constructs, only regular methods. Therefore, the filter approach is closer to the

object-oriented paradigm. Furthermore, the filters can be dynamically reconfigured (added, removed, etc.) and are able to exploit introspection provided by the underlying language.

The FLO-language [7] introduces a new component "connector" that is placed between interacting objects. The connectors are controlled through a set of interaction rules that are realized by operators (not normal methods). This connector-approach also concentrates on the messages of the objects but introduces the connectors as new entities. FLO is open for change (using a meta-object-protocol) and because the connectors are represented as objects it is close to the object-oriented paradigm.

The introduced operators are not object-oriented by nature and, therefore, less intuitive in an object-oriented system than method invocation. The approach of FLO involves a more complicated design, because in addition to the design patterns, connector objects have to be defined. The filter approach can avoid this problem by the automatic registration of filters on the involved classes.

Both mentioned approaches do not seem to offer the same ease as the filter in specializing an abstract pattern (like in [10]) to a concrete, more domain specific pattern (in the sense of [30]). Where the filters can simply use inheritance both approaches need the definition of a new domain specific layer or connector.

Hedin [12] presents an approach based on an attribute grammar in a special comment marking the pattern in the source code. This addresses the problem of traceability. The comments assign roles to the classes, which constrain them by rules like *"A DECORATOR must be a subclass of COMPONENT"*. The system can test automatically (in the source code) if the realized pattern satisfies the given and derived constraint rules.

This approach is not based on message exchanges (and is, therefore, rather simplistic), but it may be applied in any object-oriented language. It is only descriptive and not constructive (and, therefore, not reusable); each pattern must be commented again if it is applied to new application. The ability to assign constraints to patterns is interesting, especially because XOTcl provides similar abilities as well. The assertions can constrain classes (and objects) formally and informally. Both the consistency of the pattern class and its instances can be checked at run-time.

5 Conclusion

The intention of this paper is to show that object-oriented scripting languages and the management of complexity are not contradictory and that it is possible to handle complexity with a different set of advantages and tradeoffs than in "systems programming languages". Scripting is based upon several principles of programming, like using dynamic typing, flexible glueing of preexisting components, using component frameworks etc., that can lead towards a higher productivity and software reuse. We have introduced a new language construct, the filter, that offers a powerful means for the management of complex systems in a dynamic and introspective fashion. It would have been substantially more difficult to implement dynamic and introspective filters in a systems programming language. We believe that both scripting and object-orientation offer extremely useful concepts for a certain set of applications and that our approach is a useful and natural way to combine them properly.

XOTcl is available for evaluation from: `http://nestroy.wi-inf.uni-essen.de/xotcl/`

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa: *Abstracting Object Interactions Using Composition Filters*, ECOOP '93, 1993.

[2] H. Abelson, G.J. Sussman, J. Sussman: *Structure and Interpretation of Computer Programs*, MIT Press 1996.

[3] J. Bosch: *Design Patterns as Language Constructs*, http://bilbo.ide.hk-r.se:8080/~bosch/, 1996.

[4] J. Bosch: *Design Patterns and Frameworks: On the Issue of Language Support*, Workshop on Language Support for Design Pattern Frameworks at ECOOP'97, Jyväskylä 1997.

[5] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon: *Common Lisp Object System. In: Common Lisp the Language, 2nd Edition*, http://info.cs.pub.ro/onl/lisp/clm/node260.html, 1989.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented Software Architecture – A System of Patterns*, J. Wiley and Sons Ltd. 1996.

[7] S. Ducasse: *Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation*, Workshop on Language Support for Design Pattern Frameworks at ECOOP'97, Jyväskylä 1997.

[8] A.P. Ershov: *On the Essence of Compilation*, in: E. Neuhold (ed.), IFIP Working Conference on Formal Descriptions of Programming Concepts, North-Holland, New York 1978.

[9] J.L. Fontain: *Simple Tcl Only Object Oriented Programming*, http://www.mygale.org/04/jfontain/, 1998.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994.

[11] L. Hatton: *Does OO Sync with How We Think?*, in: IEEE Software, Vol. 15 (3), 1998.

[12] G. Hedin: *Language Support for Design Patterns using Attribute Extension*, Workshop on Language Support for Design Pattern Frameworks at ECOOP'97, Jyväskylä 1997.

[13] G. Kiczales, J. des Rivieres, D.G. Bobrow: *The Art of the Metaobject Protocol*, MIT Press 1991.

[14] G.E. Krasner, S.T. Pope: *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Journal of Object Oriented Programming, Vol. 1 (3), pp. 26-49, 1988.

[15] M.J. McLennan: *The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More*, Proceedings of the Tcl/Tk Workshop '95, Toronto 1995.

[16] IONA Technologies Ltd.: *The Orbix Architecture*, August 1993.

[17] H. Lieberman: *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proceedings OOPSLA '86, 1986.

[18] B. Meyer: *Object-Oriented Software Construction - Second Edition*, Prentice Hall 1997.

[19] B. Meyer: *Building bug-free O-O software: An introduction to Design by Contract*, http://eiffel.com/doc/manuals/technology/contract/index.html, 1998.

[20] G. Neumann: *Meta-Programmierung und Prolog*, Addison-Wesley 1988.

[21] G. Neumann: *A simple Transformation from Prolog-written Metalevel Interpreters into Compilers and its Implementation*, Lecture Notes in Artificial Intelligence 592, Springer, Berlin 1992.

[22] G. Neumann: *Interpretational Abstraction*, in: Computers and Mathematics with Applications, Pergamon Press, Vol. 21, No. 8, 1991.

[23] G. Neumann, S. Nusser: *Wafe – An X Toolkit Based Frontend for Application Programs in Various Programming Languages*, USENIX Winter 1993 Technical Conference, San Diego, California, January 1993.

[24] G. Neumann, U.Zdun: XOTcl, *an Object-Oriented Scripting Language*, *submitted*, 1998.

[25] J. Ousterhout: *Tcl: An embeddable Command Language*, Proceedings of the 1990 Winter USENIX Conference, 1990.

[26] J. Ousterhout: *Scripting: Higher Level Programming for the 21st Century*, in: IEEE Computer, Vol. 31, No. 3, March 1998.

[27] I. Piumarta: *SSP Chains – from mobile objects to mobile computing (Position Paper)*, ECOOP Workshop on Mobility, , Aarhus 1995.

[28] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley 1995.

[29] J. Smith, D. Smith: *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems, 2:2, June 1977.

[30] J. Soukup: *Implementing Patterns*, in: J.O. Coplien, D.C. Schmidt (Eds.), *Pattern Languages of Program Design*, Addison-Wesley 1995, pp 395-412, 1995.

[31] P. Wegner: *Learning the Language*, in: Byte, Vol. 14, No. 3, pp. 245-253, March 1989.

[32] D. Wetherall, C.J. Lindblad: *Extending Tcl for Dynamic Object-Oriented Programming*, Proceedings of the Tcl/Tk Workshop '95, Toronto, July 1995.

[33] U. Zdun: *Entwicklung und Implementierung von Ansätzen, wie Entwurfsmustern, Namenräumen und Zusicherungen, zur Entwicklung von komplexen Systemen in einer objektorientierten Skriptsprache*, Diplomarbeit (diploma thesis), Universität GH Essen, August 1998.