XOTcl– an Object-Oriented Scripting Language

Gustaf Neumann
Department of Information Systems
Vienna University of Economics and BA
Austria
gustaf.neumann@wu-wien.ac.at

Uwe Zdun
Specification of Software Systems
University of Essen
Germany
uwe.zdun@uni-essen.de

Abstract

This paper describes the object-oriented scripting language XOTcl (*Extended* OTcl), which is a value added replacement of OTcl. OTcl implements dynamic and introspective language support for object-orientation on top of Tcl. XOTcl includes the functionality of OTcl but focuses on the construction, management, and adaptation of complex systems.

In order to combine the benefits of scripting languages with advanced object-oriented techniques, we extended OTcl in various ways: We developed the filter as a powerful adapation technique and an intuitive means for the instantiation of large program structures. In order to enable objects to access several addition-classes we improved the flexibility of mixin methods by enhancing the object model with per-object mixins. We integrated the object system with the Tcl namespace concept to provide nested classes and dynamic object aggregations. Moreover, we introduced assertions and meta-data to improve reliability and self-documentation.

1 Introduction

XOTcl (pronounced exotickle) is an object-oriented scripting language providing several improvements targeted at the development and management of large systems. The base of our work was the OTcl, which is a Tcl extension introducing a dynamic object and class model by using solely the C-API of Tcl. XOTcl is a standard Tcl extension which can be dynamically loaded into every Tcl compliant environment (such as `tclsh`, `wish` or Wafe [23]).

A central property of scripting languages is the use of strings as the only representation of data. For that reason a scripting language offers a dynamic type system with automatic conversion. All integrated components (application specific extensions typically written in C) use the same string interface for argument passing. Therefore these components automatically fit together and can be reused in unpredicted situations without change. In [27] and [25] it is pointed out that *component frameworks* have proven to provide a high degree of code reuse, and are well suited for rapid application development. It is argued that application developers may concentrate primarily on the application task, rather than investing efforts in fitting components together. Therefore, in many applications scripting languages are very useful for a fast and high-quality development of software. Hatton [16] points out that the use of object-orientation in languages like C++ does not fit the human reasoning process very well. In [25] we argue that the identified deficiencies do not apply at the same degree on object-oriented scripting languages.

Tcl is equipped with functionalities like dynamic typing, dynamic extensibility and read/write introspection, that ease the glueing process of constructing systems from components. OTcl extends these important features of Tcl by offering object-orientation with encapsulation of data and operations, single and multiple inheritance, a three level class system based on meta-classes, and method chaining. Instead of a protection mechanism OTcl provides rich read/write introspection facilities, which allow one to change all relationships dynamically.

We continued and extended the design philosophy of Tcl and OTcl of providing freedom rather than constraints for the programmer. Examples are the support of dynamic changes and introspection mechanisms wherever possible. This design philosophy trades in expressiveness for protection in order to ease programming. However, a highly flexible language design implies less hard-wired protection against bad software architectures or bad programming style. We believe that no protection mechanisms can enforce the generation of coherent code/designs, so we focused on expressiveness by providing the programmer with more powerful constructs rather than on making decisions in her/his place.

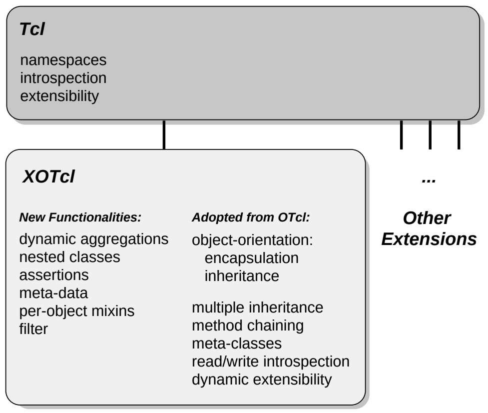

Figure 1: Language Extensions of XOTcl

The properties of OTcl described above provide a good basis for our work (see Figure 1). In the language design of XOTcl we focus on mechanisms to manage the complexity in large object-oriented systems, especially when these systems have to be adapted for certain purposes. Such situations occur frequently in the context of scripting languages. In particular we added the following support:

- *Filters* as a means of abstractions over method invocations to implement large structures, like design patterns, and to trace/adapt messages.
- *Per-object mixins*, as a means to give an object access to several different supplemental classes.
- *Dynamic Object Aggregations*, to provide dynamic aggregations through nested namespaces.
- *Nested Classes*, to reduce the interference of independently developed program structures.
- *Assertions*, to reduce the interface and the reliability problems caused by dynamic typing.
- *Meta-data*, to enhance self-documentation.

In this paper we describe theses functionalities from a language point of view. The implemented extensions provide additional functionality and lead to an improved performance in comparison to OTcl. However, we had to introduce a few incompatibilities to OTcl (discussed in Section 2 and in [33]). Since Wetherall and Lindblad provide in [32] a detailed presentation of OTcl and its design considerations, we focus here on the differences to XOTcl. The later sections introduce the new language constructs of XOTcl and discuss their usage. Finally we present a part of a larger application example (an XML-parser/-interpreter) based on design patterns, which is implemented using the new language constructs.

2 Language Constructs Derived from MIT Object Tcl (OTcl)

The `Object` command is used to create new objects. It provides access to the `Object` class which holds the common features of all objects. Objects are always instances of classes, but since objects from the most general class `Object` have no user-defined type, they may be referred to as *singular objects*. Every object can be dynamically refined with variables and with object-specific methods (using the `proc` instance method) at run-time. In the body of a proc, the predefined command `self` is used to determine the name of the current object. `self` can be used to obtain the following information about the current invocation:

- `self` (without parameters) returns the name of the currently executing object.
- `self class` returns the name of the class, which holds the currently executing method. Note, that it may differ from the object's class.
- `self proc` returns the name of the currently executing method.

A reader with OTcl knowledge will note, that there is a difference to the realization of these object informations to OTcl. XOTcl uses commands to obtain this information, whereas OTcl uses three implicit variables for this purpose (`self`, `class`, and `proc`). This change makes the internal calling conventions of XOTcl methods compatible with Tcl procedures. This has the advantage that the methods are accessible in XOTcl via namespace-paths (see Section 5). For compatibility XOTcl provides the compilation option `AUTOVARS` to set these variables automatically (with a slight performance disadvantage).

Every object is associated with a class over the `class` relationship. Classes are special objects with the purpose of managing other objects. "Managing" means that a class controls the creation and destruction of its instances and that it contains a repository of methods ("instprocs") accessible for the instances.

The instance methods common to all objects are defined in the root class `Object` (predefined or user-defined). Since a class is a special (managing) kind of object it is managed itself by a special class called "meta-class" (which manages itself). One interesting aspect of meta-classes is that by providing a constructor pre-configured classes can be created. New user-defined meta-classes can be derived from the predefined meta-class `Class` in order to restrict or enhance the abilities of the classes that they manage. Therefore meta-classes can be used to instantiate large program structures, like some design patterns. The meta-class may hold the generic parts of the structures. Since a meta-class is an entity of the programming language, it is possible to collect these in (customizable) pattern libraries for later reuse (see Section 7 for example or [25] for more details).

XOTcl supports single and multiple inheritance. Classes are ordered by the relationship `superclass` in a directed acyclic graph. The root of the class hierarchy is the class `Object`. A single object can be instantiated directly from this class. An inherent problem of multiple inheritance is the problem of name resolution, e.g. when two superclasses contain a method with the same name. XOTcl provides an intuitive and unambiguous approach for name resolution by defining the precedence order along a "*next-path*" for linearization of class and mixin hierarchies (see [32, 24] for details), which is modeled after CLOS [4]). A method can invoke explicitly the shadowed methods by the predefined command `next`. It mixes the next shadowed method on the next-path into the execution of the current method.

The usage of `next` in XOTcl is different to OTcl: In OTcl it is always necessary to provide the full argument list for every invocation explicitly. In XOTcl, a call of `next` without arguments can be used to call the shadowed methods with the same arguments (which is the most common case). When arguments should be changed for the shadowed methods, they must be provided explicitly. In the rare case that the shadowed method should receive no argument, the flag `--noArgs` must be used.

An important feature of all XOTcl objects is the read/write introspection. The reading introspection abilities are packed compactly into the `info` instance method which is available for objects and classes. All obtained information can be changed at run-time with immediate effect dynamically. Unlike languages with a static class concept, XOTcl supports dynamic class/superclass relationships. At any time the class graph may be changed entirely using the `superclass` method, or an object may change its class through the `class` method. This feature can be used for an implementation of a life-cycle or other intrinsic changes [20] of object properties (in contrast to extrinsic properties e.g. modeled through roles [14, 20] and implemented through per-object mixins [24]) . These changes can be achieved without loosing the object's identity, its inner state and its per-object behavior (procs and per-object mixins).

3 Filters

The filter is a novel approach to manage large program structures. It is a very general interception mechanism which can be used in various application areas. We have studied the use of filters for design patters in detail (see [25] and Section 7). Other useful application areas are monitoring of running systems (tracing and debugging), adaptation at runtime, implementation of proxy services, etc.

Definition 1 *A filter is a special instance method registered for a class C. Whenever an object of class C receives a message, the registered filter is invoked instead of the object's methods. The filter may handle this message and/or can decide to forward it to the object's methods.*

Usage of Filters In order to define a filter two steps are necessary: an filter-`instproc` has to be defined and the filter has to be registered using the `filter` instance method. This registration tells XOTcl, which instprocs are filters on which classes. Every filter consists of three (optional) parts:

```
ClassName instproc FilterName args {
  pre-part
  next
  post-part
}
```

The distinction into three parts is just a naming convention for explanation purposes. The pre- and post-part may be filled with any XOTcl-statements. In general the filter is free in what it does with the message. In particular it can (a) pass the message through (using the `next`-primitive), it can (b) redirect it to another destination, or it can (c) decide to handle the message itself (see Figure 2).

When a filter instproc is executed at first the instructions in the *pre-part* are processed. Then the filter might call the actual method through `next`. The filter can take the result of the actual method (returned by the next-call) and can modify it. After the execution of "next" the *post-part* is executed. Finally the caller receives the result of the filter instead of the result of the called method.

As an example we define a class `Room`. Every time an arbitrary action occurs on a room instance, the graphical sub-system should change the display of that particular room. A filter can handle the necessary notifications (here only output messages):

```
Class Room
Room instproc observationFilter args {
  puts "room action begins"
  set result [next]
  puts "room action ends -- Result: $result"
  return $result
}
Room filter observationFilter
```

When the filter is registered (last line) every action performed on an instance of `Room` is noticed with a pre- and a post-message to the standard output stream. We return the result of the actual called method, since we don't want to change the program behavior at all. When for example an instance variable is set on the instance of `Room` `r1`:

```
r1 set name "room 1"
```

the output is:

```
room action begins
room action ends -- Result: room 1
```

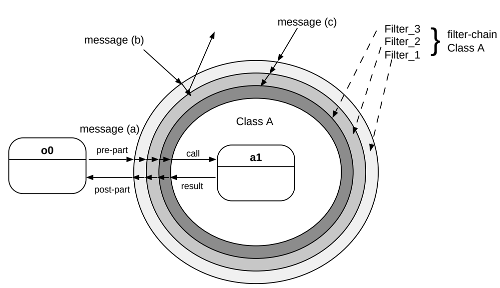

Figure 2: Cascaded Message Filtering

Filter Chains Each class may have a chain of filters which are cascaded through `next` (see Figure 2). The `next` method is responsible for the forwarding of messages to the remaining filters in the chain one by one (in registration order) until all pre-parts are executed. Afterwards the actual method is invoked and finally the post-parts are processed. If a next-call is omitted the filter chain ends in this filter method. In the following example two filters are registered, one for observation purposes and one for counting calls.

```
Room set callCounter 0  ;# set class variable
Room instproc counterFilter args {
  incr [self class]::callCounter
  next
}
Room filter {counterFilter observationFilter}
```

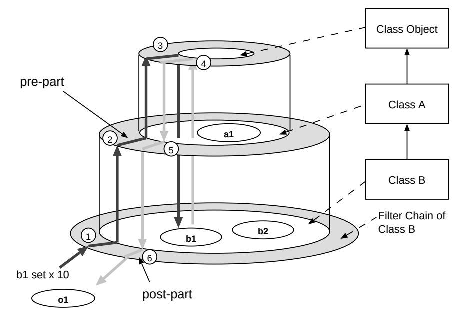

Figure 3: Filter Inheritance

Filter chains can also be combined through (multiple) inheritance using the `next` method. Filter chains of the superclasses are invoked using the same precedence order as for inheritance (see Figure 3). Without sophisticated efforts a powerful tracing facility can be implemented. E.g. a filter solely for offices distinguishes `Office`-rooms from other rooms:

```
Class Office -superclass Room
Office instproc officeFilter args {
  puts "actions in an office"
  next
}
Office filter officeFilter
```

A simple call to an instance `o1` of class `Office`, like:

```
o1 set name "office 1"
```

increments the counter and produces the output:

```
actions in an office
room action begins
room action ends -- Result: office 1
```

Introspection of Filters Filters are ordinary instprocs and have therefore access to all XOTcl functionalities including the full introspection facilities. Furthermore, filters require per-call information for reasoning or delegation purposes, i.e. information about the caller's and the callee's environment and the invocation record is required. The following options are additionally available:

- *objName* `info calledproc`: Returns the originally invoked proc.

- *objName* `info calledclass`: Returns the (presumably) called class.
- *objName* `info callingclass`: Returns the class from which the filtered call was invoked.
- *objName* `info callingproc`: Returns the proc from which the filtered call was invoked.
- *objName* `info callingobject`: Returns the object from which the filtered call was invoked.
- *objName* `info regclass`: Returns the class on which the filter is registered.
- *ClassName* `info filters`: Returns the list of filters registered for a class.

These methods return empty strings, when the desired information does not exist. The options with the prefix `calling` represent the values of `self`, `self proc`, and `self class` in the invoking method.

Tracing This example primarily demonstrates the inheritance of filter chains. Since all classes inherit from `Object`, a filter on this class is applied on all messages to objects. So all invocations of methods in the whole system are traced. The actual filter method displays the calls and exits of methods with an according message. The CALL traces are supplied with the arguments, the EXIT traces contain the result values. We have to avoid the tracing of the trace methods explicitly. With a more sophisticated filter implementation, the trace can be restricted to instances of certain classes, or produce trace output for only certain methods.

```
Object instproc traceFilter args {
  # don't trace the Trace object
  if {[self] == "::Trace"} {return [next]}
  ::set method [[self] info calledproc]
  puts "CALL > [self]->$method $args"
  ::set result [next]
  puts "EXIT > [self]->$method ($result)"
  return $result
}
Object filter traceFilter
```

Related Work The underlying idea behind filters (and per-object mixins) are interceptors for messages. In [1] objects that are able to abstract interactions among objects are introduced. The message passing model is enhanced by input/output interception, an idea generally introduced in CLOS [4].

Our driving motivation for the implementation of filters was language support for design patterns [25]. Several authors proposed other reflection and interception mechanisms for their implementation of design patterns. The LayOM-approach [5] is the most similar to the filter approach. It offers an explicit representation of 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. The filter approach differs from LayOM since it can represent patterns as ordinary classes and needs no new constructs, only regular methods. The FLO-language [11] introduces a "component connector" that is placed between interacting objects. Connectors are controlled through a set of interaction rules that are realized by operators. Hedin [17] presents an approach based on an attribute grammar in a special comment marking the pattern in the source code. The comments assign roles to the classes, which constrain them by rules. Constraining of patterns can be achieved in XOTcl using assertions (see Section 6), which can be checked at run-time.

4 Per-Object Mixins

Per-object mixins are a novel approach of XOTcl to extend the method chaining of a single object. Therefore, per-object mixins can handle complex data-structures dynamically on a per-object basis. The term "mixin" is a short form for "mixin class".

Definition 2 *A per-object mixin is a class which is mixed into the precedence order of an object in front of the precedence order implied by the class hierarchy.*

An arbitrary class can be registered as a per-object mixin for an object by the predefined `mixin` method. This method accepts a list of per-object mixins for the registration of multiple mixins. The following example defines the classes `Agent` and `MovementLog` (each with a same-named method) and registers `MovementLog` on the `Agent`-instance `a` as a mixin:

```
Class Agent
Agent instproc moveAgent {x y z} {
  puts "moving"
  # do the movement ...
}
Class MovementLog
MovementLog instproc moveAgent {x y z} {
  puts "Agent [self] moves to ($x,$y,$z)"
  next
}
Agent a -mixin MovementLog
```

Per-object mixins use the `next`-primitive to forward messages to the chain of other mixins and to pass it finally to the ordinary class hierarchy of the object. If a call on object `a` is invoked, like "`a moveAgent 1 2 3`", the per-object mixin is mixed into the precedence order of the object, immediately in front of the precedence order resulting from the class hierarchy (as illustrated in Figure 4). The resulting output of the example call is:

```
moving
Agent a moves to (1,2,3)
```

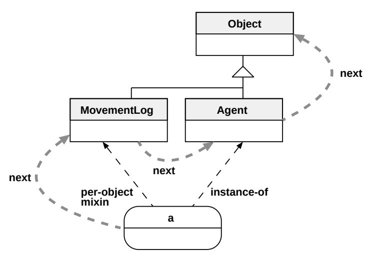

Figure 4: Per-Object Mixin Example

Mixins can be removed dynamically at arbitrary times by handing the `mixin` method an empty list. Methods of mixins have full access to all introspection options. As interceptors they additionally have access to the `info`-options `callingproc`, `callingclass` and `callingobject` (see Section 3). The registered mixins can be introspected using

objName `info mixins` *?class?*

which returns the list of all mixins of the object, when `class` is not specified. Otherwise it returns `1`, if `class` is a mixin of the object, or `0` if not.

The `mixin` relationship of an object can be used to model extrinsic properties (such as roles), whereas the `class` relationship is used to define its intrinsic properties. Per-object mixins are ordinary classes and support full specialization/generalization to make their usage compatible with ordinary classes.

Related Work Per-object mixins use the same mechanism as the method chaining in OTcl [32]. Both the precedence order and the idea of mixins in OTcl are influenced by the lisp extension CLOS [4, 18]. The filter approach (discussed in last section) is the class-level interception construct whereas per-object mixins are interceptors for single objects independent of the class relationship. A comparison between per-object mixin and filter can be found in [26]. In Agora [30] mixins are treated as named attributes of classes. In [7] different inheritance mechanisms are compared and mixins are proposed as a general inheritance construct.

Modeling of objects changing roles (as in [14]) can be implemented through the change of class relationships (see Section 2). In [24] we provide a deeper discussion of per-objects mixins and we point out that the per-object mixin are well-suited to model roles. Per-object mixins are transparent to clients. They let us decompose extrinsic (role) and intrinsic properties of objects into classes and combine them into one conceptual entity. Bosch proposes in [6] a component adaption technique, which is similar to the per-object mixin idea. It is also transparent, composable and reusable, but it is not introspective, not dynamic and a pure black-box approach.

5 Nesting of Classes and Objects

Most object-oriented analysis and design methods are based on the concepts of generalization and aggregation [29]. Generalization is achieved through class hierarchies and inheritance, while (static) aggregation is provided through embedding.

In order to support (static and dynamic) aggregation we use the namespace concept provided by Tcl since version 8.0. A *namespace* provides an encapsulation of variable and procedure names in order to prevent unwanted name collisions with other system components. Each namespace has a unique identifier which becomes part of the fully qualified variable and procedure names. Namespaces are therefore already object-based in the terminology of Wegner [31]. OTcl is object-oriented since it offers classes and class inheritance. Since objects in OTcl provide namespaces (with different semantics) as well, two incompatible namespace concepts existed in parallel. In OTcl every object has a global identifier.

XOTcl combines the namespace concept of Tcl with the object concept of OTcl. Every object and every class in XOTcl is implemented as a separate Tcl namespace. The biggest benefit of this design decision aside from performance advantages is the ability to construct aggregated objects/nested classes and to reduce name conflicts. Note, that the namespaces do not eliminate all possible naming conflicts. In XOTcl object identifiers are Tcl commands. In the case of nested objects, a name conflict between the object names and per-object procs may arise.

Through the strong integration with the Tcl namespaces we achieved additional advantages: Instance variables are traceable in XOTcl through Tcl's `trace` command, and methods may be executed via namespace qualification directly (by-passing XOTcl's dispatch), which can make method invocation as fast as Tcl's proc invocation for performance critical sections (although loosing the assertion and interception facilities of XOTcl).

Nested Classes As a simple example of nested classes, the description of a oval carpet and a desk can nest inside a `OvalOffice`-class:

```
Class OvalOffice
Class Carpet              ;# a general carpet
Class OvalOffice::Desk
# special oval carpet - no name collision
Class OvalOffice::Carpet -superclass ::Carpet
```

Nested classes have the same properties as ordinary classes. Additionally the information aboutthe nesting is available through the `info` method:

> *ClassName* `info classchildren`
> *ClassName* `info classparent`

The `classchildren` option returns a list of children (possibly empty). `classparent` results in the name of the parent class, if the class is nested. In order to ease the construction of the full path of a namespace we support the following two alternative syntax forms for the creation of nested classes:

> `MetaClassName` *ClassName*::*nestedClass*
> *ClassName* `MetaClassName` *nestedClass*

Dynamic Object Aggregation Every object in XOTcl has its own namespace which can contain other objects. Suppose an object of the class `Agent` should aggregate some property objects of an agent, such as head and body:

```
Class Agent
Class Agent::Head
Class Agent::Body
```

The classes `Head` and `Body` are in the `Agent` namespace and do not infer with other same-named classes.

```
Agent myAgent
Agent::Head ::myAgent::myHead
Agent::Body ::myAgent::myBody
```

Now `myHead` and `myBody` are part of `myAgent` and they are accessible through the full namespace path. These paths can turn out to be quite cumbersome to write. Fortunately, in most situations a programmer does not have to write the full path, since within XOTcl methods the object's namespace is set automatically and the `self` command obtains the fully qualified object name. For the creation of aggregated objects the following two forms can be used:

> `ClassName` *objName*::*aggregatedObj*
> *objName* `ClassName` *aggregatedObj*

The information about the part-of relationship of objects can be obtained the same way as for classes through the `info` method interface:

> *objName* `info children`
> *objName* `info parent`

Dynamic Aggregation It is likely that all agents have properties for head and body. This implies a static or pre-determined relationship between class nesting and object aggregation. A pre-determined aggregation of property objects can be built through a constructor, such as:

```
Agent instproc init args {
  ::Agent::Head [self]::myHead
  ::Agent::Body [self]::myBody
}
Agent myAgent
```

Every agent is now created with a head and a body. The aggregation can be changed dynamically at runtime by creation or destruction of objects. The `destroy` method turns the agent into a headless agent:

```
myAgent::myHead destroy
```

XOTcl provides introspection for aggregations as well. Suppose, that in the virtual world the agents heads may be slashed from their bodies. The graphical system can simply ask the agent with `info children`, whether it has a head or not, and can choose the graphical representation accordingly.

Every object/class can be moved (and copied) to an other object/class by the `move` (`copy`) method. These are deep operations, effecting the object and all its aggregates.

Related Work Our view of aggregation is influenced by investigations on modules [13], conceptual modeling [29], and object-oriented languages, like Troll [15]. Several common programming languages offer nested or inner classes, e.g. Java, C++, Beta, etc. These concepts provide aggregation of descriptive structures, but lack in introspection abilities and dynamics. Banavar [3] points out that class-level nesting is a form of composition.

In languages without support for object aggregation, it is approximated by embedding of objects or by association through pointers (a reference). Embedding does not permit dynamic aggregations, pointers lead to low level programming with hand coded operations for operators like deep-copy/-move. These approximations contradict the idea of an aggregation.

Several design patterns also use aggregations for composition. The whole-part pattern [9] aggregates objects of arbitrary types. Dynamic object aggregations form a language support for this pattern. The more special variant "composite" [12] aggregates hierarchies of objects of the same type and can also be language supported (see Section 7).

6 Other Functionalities

This section describes briefly some other language functionalities that are not available in OTcl.

Abstract Classes A class is defined abstract if at least one method of this class is abstract. The build-in method `abstract` defines an specifies the interface of an abstract method. Direct calls to abstract methods produce an error message. E.g. a `Graphic`-class provides an abstract interface for drawing:

```
Class Graphic
Graphic abstract instproc draw args
```

Parameters Classes may be equipped with `parameters` definitions which are automatically created for the convenient setting and querying of instance variables. Parameters may have a default value, e.g.:

```
Class Person -parameters {
  name
  {friends ""}
  {ID [self]}
}
```

Each instance of class `Person` has three properties defined. `name` has no default value, `friends` defaults to an empty list, and the `ID` defaults to the instance's self-ID. `parameters` are inherited to subclasses. The following example demonstrates setting and querying of parameters:

```
Person p1 -name Anakin ;#set name at creation
p1 name "Darth Vader"  ;#set name at runtime
puts "Name of p1: [p1 name] objID: [p1 ID]"
```

Assertions In order to improve reliability and self documentation we added assertions to XOTcl. The implemented assertions are modeled after the "design by contract" concept of Bertrand Meyer [21, 22]. In XOTcl assertions can be specified in form of formal and informal pre- and post-conditions for each method. The conditions are defined as a list of and-combined constraints. The formal conditions have the form of ordinary Tcl conditions, while the informal conditions are defined as comments (specified with a starting "`#`"). Pre- and post-conditions are appended as lists to the method definition.

Since XOTcl offers per-object specialization it is desirable to specify conditions within objects as well (this is different to the concept in [21]). Furthermore there may be conditions which must be valid for the whole class or object at any visible state (that means in every pre- and post-condition). These are called invariants. Logically all invariants are appended to the pre- and post-conditions with a logical "and". The syntax for class invariants is:

> *ClassName* `instinvar` *invariantList*

and for objects invariants:

> *objName* `invar` *invariantList*

All assertions may be introspected. Since assertions are contracts they need not to be tested if one can be sure that the contracts are fulfilled by the partners (see [21]). But for example when a component has changed or a new one is developed the assertions could be checked on demand. The checking is then fulfilled at the beginning and at the end of each method call. The `check` method has configuration options for all assertions types to turn checking on/off. The syntax is:

> *objName* check ?all? ?instinvar? ?invar? ?pre? ?post?

Per default all options are turned off. `check all` turns all assertion options for an object on, an arbitrary list (maybe empty) can be used for the selection of certain options. Assertion options are introspected by the `info check` option. The following class is equipped with assertions:

```
Class Sensor -parameters {{value 1}}
Sensor instinvar {
  {[regexp {^[0-9]$} [[self] set value]] == 1}
}
```

```
Sensor instproc incrValue {} {
  incr [self]::value
} \
{{# pre-condition:} {[[self] value] > 0}}  \
{{# post-condition:} {[[self] value] > 1}}
```

The `parameter` instance method defines an instance variable `value` with value `1`. The invariant expresses the condition (using the Tcl command `regexp`), that the value must be a single decimal digit. The method definition expresses the formal contract between the class and its clients that the method `incrValue` only gets input-states in which the value of the variable `value` is positive. If this contract is fulfilled by the client, the class commits itself to supply a post-condition where the variable's value is larger than 1. The formal conditions are ordinary Tcl conditions. If checking is turned on for sensor `s`:

```
s check all
```

the pre-conditions and invariants are tested at the beginning and the post-condition and invariants are tested at the end of the method execution automatically. A broken assertion, like calling `incrValue` 9 times (would break the invariant of being a single digit) results in an error message.

We have already pointed out that the presented concepts are relying on Meyer's Design by Contract [21, 22]. The differences are, that due to the ability to define per-object specializations object assertions are introduced, and that due to the dynamics of the language the assertions are also dynamically changeable and introspectable.

AsserTcl [10] is another approach that introduces assertions into Tcl through four new Tcl commands. The advantages of our approach are the integration with object-orientation , the placement of assertions at a familiar place (at the end of the method definition) and the support for invariants in classes and objects.

Meta-data To enhance the self-documentation and the consistency between documentation and program it is useful to make the documentation a part of the program, i.e. to store meta-data like the author, a description, the version, etc. Meta-data registered for classes is inherited and propagated to all instances and are a dynamic and introspective. Syntactically, meta-data can be specified through the `metadata` method with its options `add` and `remove`. Other arguments for the `metadata` method are interpreted as meta-data values. E.g. on `Object`:

```
Object metadata add {description version}
Object metadata description "This class \
  realizes all common object behavior"
Object metadata version "1.0"
```

Since the meta-data registered on classes is inherited, all objects can store information on `description` and `version`. E.g. the agent `a1` stores such values:

```
Agent a1
a1 metadata description "My testing agent"
a1 metadata version "0.1"
```

Classes and objects can store additional meta-data for their own purposes at any time. E.g. agents can store information about the host on which they have been created. Introspection of meta-data is implemented through `info metadata` method, which lists all defined meta-data attributes with values associated (only those). Meta-data values can be accessed through the `metatdata` instance method, similar to the usage of `set`. If no value parameter is given, the current value is returned. The following command produces the result "`1.0`":

```
Object metadata version
```

Beneath the special features, like inheritance, meta-data could be expressed through instance variables solely. But especially in distributed environments, it is important to have such a facility, because the common place and the introspection mechanisms allow different people and, in XOTcl's case, even other programs, to gain the meta-data without searching for them. Since this is not only a naming convention, but a language construct, the interpreter results in an error if the meta-data is used incorrectly.

Automatic Name Creation The XOTcl `autoname` instance method provides an simple way to take the task of automatically creating names out of the responsibility of the programmer. The example below show how to create on each invocation of method `new` an agent with a fresh name (prefixed with `agent`):

```
Agent proc new args {
  eval [self] [[self] autoname agent] $args
}
```

7 Application Example: An XML-Parser/-Interpreter

Now we illustrate the usage of the new language constructs in a larger example: the design pattern based architecture/implementation of an XML-Parser/-Interpreter. XML [8] is a meta-language that defines on basis of a document type definition (DTD) the structure of an application document.

An application program that wants to extract information from an XML document has to parse the document and has to interpret the resulting structure. The (partial) design of our implementation is presented in Figure 5, where a wrapper facade pattern integrates and encapsulates an off-the-shelf XML parser, the interpreter-/composite-patterns abstracts from the syntax tree representation, a builder separates the parsing from the creation of the resulting structure, a visitor decouples the interpretation from the syntax tree and a per-object observer is used to trace visitations.

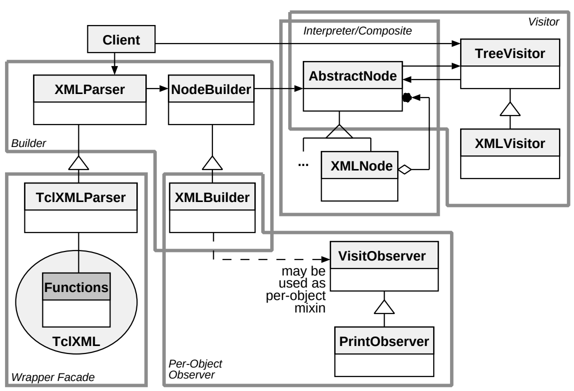

Figure 5: Partial Design of the XML Parser/Interpreter

The interpreter pattern [12] defines an object-oriented representation for a grammar along with an interpretation mechanism (essentially an interpret method for each node type). All clients abstract from expressions through the use of an abstract interface for interpretation purposes. At run-time expression objects form an abstract syntax tree. Frequently the interpreter's tree representation is implemented as a composite pattern, that arranges objects in a tree with leaf and composite nodes of the same component type. Registered composite operations are recursively forwarded to all children of the composite.

The composite pattern can be implemented through a meta-class that registers a filter in the constructor of the meta-class for forwarding of the operations to the nested nodes. We implement the abstract pattern in a meta-class `Composite` which can be loaded from a library and reused and instantiated in several application classes. `addOperations` registers and `removeOperations` unregisters the operations which should be forwarded to the nested nodes.

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

The registered operations are stored in the associative array `ops` (a class variable in the class on which the filter was registered) that is accessed by a generic filter which performs the actual forwarding:

```
Composite instproc compositeFilter args {
  set m [[self] info calledproc]
  set c [[self] info regclass]
  set r [next]
  if {[info exists ${c}::ops($m)]} {
    foreach child [lsort \
      [[self] info children]] {
      eval [self]::$child $m $args
    }
  }
  return $r
}
```

The filter determines through `info calledproc` the method which is called, obtains the registration class of the filter through `info regclass`, and checks, whether the called method `m` was registered in the array `ops` for forwarding. If `m` is registered, the message is passed to all children objects (determined by `info children`). Finally, `next` forwards the message to the actual node. Since the children may be composites as well, this mechanism iterates recursively on the entire tree structure. The filter is registered for derived application classes in the constructor of `Composite`.

Now the meta-class `Composite` can be used to implement the interpreter pattern for XML-nodes. We define a class `AbstractNode` with the meta-class `Composite` and an abstract operations for accepting interpretation through a visitor, which should work recursively, therefore it is added as a composite operations. Application node classes such as `XMLNode` can be derived by specializing `AbstractNode`:

```
Composite AbstractNode
AbstractNode abstract instproc accept v
AbstractNode addOperations accept
Class XMLNode -superclass AbstractNode
```

For parsing of XML documents several parsers (such as TclXML [2]) can be used. In order to use a legacy parser in an exchangeable manner it has to be encapsulated transparently. The wrapper facade pattern [28] provides a general means to shield clients from direct dependencies to functions.

Actually we use `XMLParser` as a wrapper facade object that embodies the interface to the XML parser as object-oriented methods. `configure` sets the parser configuration, `cget` queries the configuration, `parse` invokes the parsing of XML text, and `reset` cleans up the parser context before a new text can be parsed.

```
Class XMLParser -parameters {nodeBuilder}
XMLParser abstract instproc init args
XMLParser abstract instproc cget option
XMLParser abstract instproc configure args
XMLParser abstract instproc parse data
XMLParser abstract instproc reset {}
```

A specific TclXML parser is derived by sub-classing:

```
Class TclXMLParser -superclass XMLParser
```

In order to separate the construction process of the complex node structure from its representation and to make representations exchangeable, we use the builder pattern [12]. The parser is the director of the construction process, invoked by `parse` operation. `parse` uses the `NodeBuilder` interface containing three methods to build the data representation (`startElt` actually creates nodes). A concrete implementation `XMLBuilder` builds the abstract syntax tree.

```
Class NodeBuilder
NodeBuilder abstract instproc charData text
NodeBuilder abstract instproc startElt \
  {tag attrList}
NodeBuilder abstract instproc endElt tag
Class XMLBuilder -superclass NodeBuilder
```

The visitor pattern [12] is used to define operations on the nodes independently from the intrinsic properties of the nodes. According to the pattern the nodes have to concretize an `accept` method , which calls the `visit` method of the visitor for all nodes (since it is registered as a composite method):

```
XMLNode instproc accept v {$v visit [self]}
```

Below is an abstract `TreeVisitor` and a specialization `PrintVisitor` that just prints out every node:

```
Class TreeVisitor
TreeVisitor abstract instproc visit objName
Class PrintVisitor -superclass TreeVisitor
PrintVisitor instproc visit objName {
  puts "Visitation of node $objName"
}
```

Such visitors can be used e.g. for locating XML-elements with certain properties, for extracting or mapping of the XML-structure, etc. For observing certain events in the system, a per-object observer [26] may be used, which can for example observe the parsing of the document. The per-object observer is based on the observer in [12] and implemented through per-object mixins. The `PrintObserver` watches the processing of `startElt` tags:

```
Class PrintObserver
PrintObserver instproc startElt {t a} {
  puts stderr "... watching: name=$n"; next
}
```

The `PrintObserver` can be added to the `XMLBuilder`, which should be observed. Finally, the parser is connected with the node-builder instance:

```
XMLBuilder ::t -mixin PrintObserver
TclXMLParser x -nodeBuilder ::t
x parse "...<PERSON>...</PERSON>..."
```

The implementation of the presented XML processing system has the size of 73 lines (including two more useful visitors) plus 22 lines for the wrapper facade, and 25 lines for the implementation of `Composite` (including the definitions of abstract interfaces).

8 Conclusion

This paper introduces XOTcl, which is an experiment to combine the benefits of a scripting language with the benefits of a high level object-oriented language. We tried to preserve the underlying principles of the scripting language Tcl (like dynamic typing, flexible glueing of preexisting components, read/write introspection) while extending the language with high level object-oriented concepts (like filters, per-object mixins and dynamic aggregations). Our goal was to improve productivity and software reuse by providing the programmer with powerful means to manage complexity and to improve composability.

We used the new language concepts with promising success in various applications, including a web browser [19], an HTTP server, a web-based object and a mobile code system [34], a persistent store, an XML-/RDF-Parser, etc. The new language constructs helped to improve the modularization, the robustness and the code size of these systems.

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

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. LCNS 791, Springer-Verlag, 1993.

[2] S. Ball. TclXML. http://www.zveno.com/zm.cgi/in-tclxml/, 1999.

[3] G. Banavar. Nesting as a form of composition. In *Proc. of CIOO Workshop at ECOOP*, July 1996.

[4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification. *Sigplan Notices*, 23(9), 1988.

[5] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[6] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41, 1999.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA/ECOOP'90*, volume 25 of *SIGPLAN Notices*, pages 303–311, October 1990.

[8] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language. http://www.w3.org/TR/REC-xml, 1999.

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

[10] J. Cook. Assertions for the Tcl language. In *Proc. of the Fifth Annual Tcl/Tk Workshop 1997*, Boston, 1997.

[11] S. Ducasse. Message passing abstractions as elementary bricks for design pattern implementations. In *Proc. of LSDF'97*, 1997.

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

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[14] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), 1996.

[15] T. Hartmann, R. Junghans, and G. Saake. Aggregation in a behavior oriented object model. In O. Madsen, editor, *Object-Based Distributed Processing*, pages 57–77. LCNS 615, Springer-Verlag, 1992.

[16] L. Hatton. Does OO sync with how we think? *IEEE Software*, May/June 1998.

[17] G. Hedin. Language support for design patterns using attribute extension. In *Proc. of LSDF'97*, 1997.

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

[19] E. Köppen, G. Neumann, and S. Nusser. Cineast – an extensible web browser. In *Proc. of the WebNet 1997 World Conference on WWW, Internet and Intranet*, Toronto, Canada, November 1997.

[20] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[22] B. Meyer. Building bug-free o-o software: An introduction to design by contract. http://eiffel.com/doc/manuals/technology/contract/index.html, 1998.

[23] G. Neumann and S. Nusser. Wafe – an X toolkit based frontend for application programs in various programming languages. In *Proc. of USENIX Winter 1993 Technical Conference*, San Diego, January 1993.

[24] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. Accepted for publication at Asia-Pacific Software Engineering Conference (APSEC), December 1999.

[25] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proc. of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, May 1999.

[26] G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proc. of NOSA'99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.

[27] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.

[28] D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.

[29] J. Smith and D. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), 1977.

[30] P. Steyaert, W. Codenie, T. D'Hondt, K. D. Hondt, C. Lucas, and M. V. Limberghen. Nested mixin-methods in Agora. In *Proc. of ECOOP '93*, LNCS 707. Springer-Verlag, 1993.

[31] P. Wegner. Learning the language. *Byte*, 14(3):245 – 253, March 1989.

[32] D. Wetherall and C. J. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop '95*, Toronto, July 1995.

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

[34] U. Zdun. Entwurf und Entwicklung eines mobilen Objekt-Systems für Anwendungen im Internet. Diplomarbeit (diploma thesis), Universität Gesamthochschule Essen, 1999.