Object Orientation for Tcl
(also available as TIP #257)

Donal K. Fellows <donal.k.fellows@manchester.ac.uk>
Will Duquette <will@wjduquette.com>
Steve Landers <steve@digitalsmarties.com>
Jeff Hobbs <jeffh@activestate.com>
Kevin Kenny <kennykb@users.sourceforge.net>
Miguel Sofer <mig@utdt.edu>

*This TIP proposes adding OO support to the Tcl core, semantically based on XOTcl. The commands it defines will be in the **::oo** namespace, which is not used by any current mainstream OO system, and it will be designed specifically to allow classic XOTcl to be built on top.*

1. Rationale and Basic Requirements

Tcl has a long history of being comparatively agnostic about object-oriented programming, not favouring one OO system over another while promoting a wealth of OO extensions such as [incr Tcl], OTcl, XOTcl, stooop, Snit, etc. because in general, one size fits nobody.

However, many application domains require OO systems and having a common such base system will help prevent application and library authors from reinventing the wheel each time through because they cannot rely on an OO framework being present with every Tcl installation. For example, the http package supplied with Tcl has its own internal object model, and similar mechanisms have been reinvented multiple times within tcllib. Other parts of tcllib do their own thing (to say nothing of the fact that both stooop and Snit are in tcllib themselves). This does not promote efficient reuse of each others code, and ensures that each of these packages has a poor object system. The request for an OO system is also one of the biggest feature requests for Tcl, and would make it far easier to implement megawidgets. It also leaves Tcl open to the ill-informed criticism that it does not support OO, despite being spoilt for choice in reality through the extensions listed above.

Given all this, the time has come for the core to provide OO support. The aim of the core OO system shall be that:

- it is simple to get started with,
- flexible so that it can take you a long way,
- fast (we all know that we're going to get compared on this front!), and
- suitable for use as a foundation of many other things, including the reimplementation of various existing OO extensions, including those that are currently compiled and also those that are pure Tcl extensions.

Another requirement is that programmers should not have to alter all of their existing code in order to get started with the new system; rather, they should be able to adopt it progressively, over time, because it supports betters ways of working (*e.g.*, faster and more flexible libraries).

2. The Foundational OO System

This TIP proposes that the foundation of the OO system should be based on XOTcl as that is fast, semantically rich, well supported, and relatively compatible with the existing Tcl build system.

Some changes will be necessary. Certain aspects of XOTcl syntax are peculiar from a conventional OO point-of-view, and it is deeply unfortunate that a very large number of methods are predefined in the XOTcl base class. XOTcl's approach to object creation options is also highly idiosyncratic, and does not support the typical Tcl idioms. However, the changes must be made in such that classic XOTcl can be built on the new framework; as a result, the classic XOTcl base class will be derived from something more fundamental.

2.1 Key Features

- Class-based object system. This is what most programmers expect from OO, and it is very useful for many tasks.
- Allows per-object customization and dynamic redefinition of classes.
- Supports advanced OO features, such as:

  meta-classes These are subclasses of **class**, which permit more advanced customization of class behaviour.

  filters These are constraints (implemented in Tcl code, naturally) on whether a method may be called. They may also substitute another value for the result of a method.

  mixins These allow functionality to be brought into an object from other objects if necessary, enabling better separation of concerns.

  invariants These ensure that assumptions about the behaviour of a class can be checked.

2.2 Key Alterations

- Object and class names in the core extension to be all lower-case, in line with best common practice.
- Methods have to be capable of being non-exported, by which we mean that they are not (simply) callable from contexts outside the object.
- The majority of the API for updating an object or class's definition is to be moved to a separate utility command, **oo::define**.

- More “conventional” naming of operations is to be used.

Note that this TIP does not propose to actually include any XOTcl (or [incr Tcl] or Snit or …) compatibility packages in the core; it is about forming a foundation on which they can be built (which happens to also be a comparatively lightweight OO system in itself). Such compatibility packages can either remain separate code, or be the subject of future TIPs.

3. Outline Proposal

If you do not know XOTcl, you can probably skip forward to Section 4 below.

3.1 Essential Changes Relative to XOTcl

This section describes the essential changes to XOTcl behaviour required to meet the above goals and the rationale for them. The paragraphs that describe the specific changes begin with the word **Therefore**, in bold type. Note that wherever possible, the semantics of XOTcl are to be used even where the syntax is not; deviations will be explicitly listed.

3.1.1 Exported vs. Non-exported Methods

In XOTcl, every class and every object has an associated namespace. The namespace associated with a class *::myclass* is *::xotcl::classes::myclass*; the namespace associated with object *::myobject* is simply *::myobject*. XOTcl “*instprocs*” are simply procedures defined in a class (or superclass) namespace; XOTcl per-object “*procs*” are simply procedures defined in an object’s namespace. *Every such procedure becomes an object subcommand.*

This is part of the reason why XOTcl objects have such cluttered interfaces. Every method that is of use to the object appears in the object’s interface – and there is no way to prevent this.

Therefore, in the new OO system “**proc**s” and “**instproc**s” can be exported or non-exported. Exported **proc**s appear as object subcommands; non-exported **proc**s do not, but remain available as subcommands of the **my** command. In this way, the object itself can still use them, but they need appear in the object’s interface only if desired.

Additionally, the standard **info** method will need to be extended to allow introspection of which methods are exported and which are not.

3.1.2 The oo::define Command

In XOTcl, the commands to define per-class methods, filters, and so on are subcommands of the class object; the commands to define per-object methods, filters, and so on are subcommands of the individual object. This is a problem, as it confuses the implementation-time interface with the run-time interface. The design is logical, given XOTcl’s extreme dynamism; any implementation-time activity, such as defining a method or adding a filter can indeed be done at run-time. Again, this makes it difficult to define clean run-time interfaces for reuseable library code.

The solution described in the previous section, of making some methods private by declaring them non-exported, does not give us a full solution; having the **instproc** subcommand available only from instance code is not very useful.

Therefore, we add a new command, **oo::define**, which is used to define methods, filters, and so on. It can be called in two ways. The first is as follows:

> **oo::define** *objectOrClass subcommand args...*

For example, the following XOTcl code defines a class with two methods:

```
xotcl::Class myclass
myclass instproc dothis args { # body }
myclass instproc dothat args { # body }
```

In the new OO core, the matching code would be this:

```
oo::class myclass
oo::define myclass instproc dothis args { # body }
oo::define myclass instproc dothat args { # body }
```

oo::define can also be called with a script whose commands are aliased to the subcommands of **oo::define**. Thus, the above code could also be written as follows:

```
oo::class myclass
oo::define myclass {
    instproc dothis args { # body }
    instproc dothat args { # body }
}
```

Finally, the class "*create*" method could be extended so that it could be called with such a script:

```
oo::class myclass {
    instproc dothis args { # body }
    instproc dothat args { # body }
}
```

This allows a class to be defined cleanly and concisely, while guaranteeing that all class details can still be modified later on using **oo::define**.

Note that we do not lose any object-oriented flexibility by this scheme. An *oo::xotcl* package can use the "**forward**" feature to forward "**instproc**" and its partners to **oo::define**, thus defining them all as methods; and once they are methods, all of the usual techniques of method chaining, mix-ins, and filters apply.

oo::define will need two subcommands XOTcl does not currently provide: **export** and **unexport**. **Export** takes as arguments a list of method names; all named methods are exported and become visible in the object or class's interface. **Unexport** does the opposite. Each can include wildcards in its argument list, just as **namespace export** does.

3.1.3 Standard Metaclasses

XOTcl defines two standard Metaclasses, *xotcl::Object* and *xotcl::Class*. *xotcl::Object* is the root of the class hierarchy; all XOTcl classes implicitly inherit from *xotcl::Object*. XOTcl classes are themselves objects, and are instances of *xotcl::Class*. *xotcl::Class* can itself be subclassed to produce different families of classes with different standard behaviours.

The new core object system will use the same basic mechanism, based on the metaclasses **oo::object** and **oo::class**. However, one of the problems with XOTcl is that XOTcl objects have

too much standard behaviour; the new core object system must provide a simpler foundation, with the XOTcl behaviour optionally available.

Therefore, we will decompose the features of *xotcl::Object* and *xotcl::Class* into a number of simpler metaclasses.

oo::object will be the root of the class hierarchy. However, instances of **oo::object** will have a minimal set of standard methods, so that clean interfaces can be built on top of it, as can be done with Snit types and instances.

Core object system classes will be instances of **oo::class** or its subclasses. Likewise, **oo::class** will define only minimal behaviour.

The majority of standard XOTcl class and object methods will be provided by a number of standard classes, all of which will be subclasses of **oo::object**. A user-defined class can include some or all of the standard XOTcl behaviour by multiply inheriting from some or all of these standard classes. Each such standard class will provide a subset of the standard XOTcl methods. The following is an incomplete list of the necessary classes:

- **oo::definer** will define one method for each subcommand of **oo::define**; the methods will be "**forward**"ed to **oo::define**.
- **oo::struct** will define all of the data access methods, *e.g.*, **set**, **unset**, **lappend**, **incr**, and so forth.

Thus, **oo::class** is the mechanism for defining classes with clean interfaces and maximum data hiding and encapsulation; **oo::struct** is the mechanism for defining classes for maximal public access.

The above classes and metaclasses will be implemented such that they can be used as a foundation for the *::xotcl::Class* and *::xotcl::Object* metaclasses (see below for a discussion).

3.1.4 Inheritance

A class may wish to make use of the capabilities of **oo::struct** internally without exporting its methods.

Therefore, the inheritance mechanism should be extended such that the newly defined class can declare whether a parent class's methods should be exported or not.

3.1.5 Object Creation

XOTcl has a unique creation syntax. The object name can be followed by what look like Tk or Snit options – but are not. Instead, any token in the argument list that begins with a hyphen is assumed to be the name of one of the object's methods; it must be followed by the method's own arguments. For example, a standard XOTcl class will have a "*set*" method, which has the same syntax as the standard Tcl **set** command. Thus, the following code:

```
myclass myobj -set a 1 -set b 2
```

creates an instance of "*myclass*" called "*myobj*" whose instance variables "*a*" and "*b*" and set to 1 and 2 respectively. This is an intriguing and innovative interface, and it is unlike any other Tcl object system. Additionally, it makes it difficult to implement standard Tk-like options.

Therefore, standard core object system classes will not use this mechanism (though it might be available on demand by inheriting from some other standard metaclass). Instead, standard core object system classes will have no creation behaviour other than that implemented by their designers in their constructors.

Constructors may have any argument list the user pleases, including default arguments and the "**args**" argument (as in the **proc** command). It is up to the developer to handle the arguments appropriately.

It is expected that one of the key responsibilities of any XOTcl compatibility package would be to define a constructor that parses the arguments in the expected way and uses them to invoke methods on the newly created object.

3.1.6 Constructor Syntax

In XOTcl, a class's constructor is implemented using its "*init*" instproc. This is troubling; constructors are intended to do things just once, and are often written to take advantage of that, whereas an "*init*" instproc can theoretically be called at any time. For any given class, then, one of two conditions will obtain: either "*init*" must be written so that it can be called at any time, or the class will have an inherent logic bug.

Therefore, the class constructor will not be implemented as a standard instproc. Instead, the **oo::define** command will have a new subcommand, **constructor**, which will be used as follows:

```
oo::define myclass constructor {
    # body
}
```

The constructor so defined will act almost exactly like an instproc; it may have pre- and postconditions attached to it, it may call superclass constructors using the **next** command, etc. However, it may never be called explicitly, but only via the class's **create** and **new** methods.

3.1.7 Destructor Syntax

In XOTcl, a class's destructor is defined by overriding the "*destroy*" instproc. This is problematic for two reasons: first, a destructor does not need an argument list, and has no need of preconditions and postconditions. An instproc is too powerful for the task. Second, successful destruction should not depend on the destructor's chaining to its superclass destructors properly.

Therefore, the class destructor will be defined by a new subcommand of **oo::define**, **destructor**, as follows:

```
oo::define myclass destructor {
    # Body
}
```

The destructor has no argument list, nor does it have any preconditions or postconditions.

The destructor cannot be called explicitly. Instead, the destructors are invoked in the proper order by the standard **destroy** method (defined in oo::object), which need never be overridden.

3.2 Desirable Changes

The changes described in this section are not essential to meeting the goals described earlier. However, they are desirable in that they lead to cleaner, more maintainable code.

3.2.1 Class vs. Object Method Naming

XOTcl has many features which can be applied to a class for use by all class instances, or to a single object. For example, a "*filter*" can be defined for a single object, while an "*instfilter*" can be defined for a class and applied to all instances of that class.

This is exactly backward. Most behaviour will be defined for classes; additional per-object behaviour is the special case, and consequently should have the less convenient name.

Therefore, all XOTcl subcommands that begin with "*inst*" will lose their "*inst*" prefix; the matching per-object subcommands will gain a "*self.*" prefix, to indicate that it is operating on the object itself and not the members of the class. Thus, a filter is defined on a class for its instances using the "**filter**" subcommand; a filter is defined on a particular object using the "**self.filter**" subcommand.

3.2.2 Procs vs. Methods

The word "*proc*" conveys a standalone function; an object's subcommands are more typically described as its "*methods*".

Therefore, the XOTcl "*instproc*" and "*proc*" subcommands should be renamed as "*instmethod*" and "*method*", or, if the new naming convention described in the previous section is adopted, **method** and **self.method**.

3.2.3 Public Names

In XOTcl, the main objects are *xotcl::Class* and *xotcl::Object*. However, the Tcl Style Guide dictates that public command names begin with a lower-case letter.

Therefore, all public names in the **oo::** namespace will begin with a lower case letter, *e.g.*, the standard core object system equivalents of *xotcl::Class* and *xotcl::Object* will be **oo::class** and **oo::object** respectively.

The names in any *oo::xotcl* compatibility module would naturally follow the existing XOTcl conventions.

4. API Specification

This section documents the core object system API in detail, based on the essential and desirable changes discussed in the previous sections.

4.1 Helper Commands

The namespace(s) that define the following three commands are not defined in this specification; all that is defined is that they will be on the object's **namespace path** during the execution of any method and should always be used without qualification.

4.1.1 my

The **my** command allows methods of the current object to be called during the execution of a method, just as if they were invoked using the object's command. Unlike the object's command, the **my** command may also invoke non-exported methods.

my *methodName* ?*arg arg …*?

4.1.2 next

The **next** command allows methods to invoke the implementation of the method with the same name in their superclass (as determined by the normal inheritance rules; if a per-object method overrides a method defined by the object's class, then the **next** command inside the object's method implementation will invoke the class's implementation of the method). The arguments to the **next** command are the arguments to be passed to the superclass method (or the intercepted method in the case of filters and mixins); this is in contrast to the XOTcl **next** command, but other features in Tcl 8.5 make this approach viable and much easier to control. The current stack level is temporarily bypassed for the duration of the processing of the **next** command; this is in contrast to the XOTcl version of the **next** command, but it allows a method to always execute identically with respect to the main calling context.

next ?*arg arg ...*?

It is an error to invoke the **next** command when there is no superclass definition of the current method.

4.1.3 self

The **self** command allows executing methods to discover information about the object which they are currently executing in. Without arguments, the **self** command returns the current fully-qualified name of the object (to promote backward compatability). Otherwise, it is a command in the form of an ensemble (though it is not defined whether it is manipulable with **namespace ensemble**).

The following subcommands of **self** are defined. None of these subcommands take additional arguments.

caller Returns a three-item list describing the class, object and method that invoked the current method, respectively. Syntax:

self caller

class Returns the name of the class that defines the currently executing method. If the method was declared in the object instead of in the class, this returns the class of the object containing the method definition. Syntax:

self class

filter When invoked inside a filter, returns a three-item list describing the object or class for which the filter has been registered. The first element is the name of the class or object, the second element is either **method** (for a method defined in a class for its instances) or **self.method** (for a method defined by a single object), and the third element is the name of the method.

self filter

method Returns the name of the currently executing method. Syntax:

self method

namespace Returns the namespace associated with the current object. Syntax:

self namespace

next Returns the fully-qualified name of the method that will be executed when the **next** command is invoked, or an empty string if there is no superclass definition for the method. Syntax:

self next

object Returns the name of the current object, the same as if the **self** command is invoked with no arguments. Syntax:

self object

target When invoked from a filter or mixin, returns a two-item list consisting of the name of the class that holds the target method and the name of the target method. Syntax:

self target

4.2 Core Objects

The following classes are defined, and are the only pre-constructed objects in the core system.

4.2.1 oo::object

oo::object *name*

Constructs a new object called *name* of class **oo::object**; the object is represented as a command in the current scope. **oo::object** returns the fully qualified command name. If *name* is the empty

string, **oo::object** generates a name automatically that is guaranteed to not clash with any existing command name.

The name of an object is also the name of a command in the form of an ensemble where the subcommands of the ensemble are the exported method names of the object.

The new object has two predefined non-exported methods: **eval** and **variable**. Other subcommands and other behaviour can be added using **oo::define**.

oo::object serves as the base class for all other **oo::** classes.

4.2.1.1 Methods

The instances of **oo::object** (*i.e.*, all objects and classes) have the following methods:

eval This non-exported method concatenates its arguments according to the rules of **concat**, and evaluates the resulting script in the namespace associated with the object. The result of the script evaluation is the result of the *object* **eval** method.

object **eval** ?*arg arg* …?

variable This non-exported method takes an arbitrary number of *unqualified* variable names and binds the variable with that name in the object's namespace to the same name in the current scope. If an argument consists of a two-element list, the first element is the name of the variable to bind in the object's namespace, and the second element is the name of the variable to bind in the current scope.

object **variable** ?*varName varName* …?

4.2.1.2 Unknown Method Handling

When an attempt is made to invoke an unknown method on any object, the core then attempts to pass all the arguments (including the command name) to the public **unknown** method of the object. If no such method exists, an error message is generated. Instances of the core **oo::object** class do not have an unknown method by default.

4.2.2 oo::class

This class is the class of all classes (*i.e.*, its instances are objects that manufacture objects according to a standard pattern). Note that **oo::object** is an instance of **oo::class**, as is **oo::class** itself.

oo::class *name* ?*definition*?

This creates a new class called *name*; the class is an object in its own right (of class **oo::class**), and hence is represented as a command in the current scope. **oo::class** returns the fully qualified command name. If *name* is the empty string, **oo::class** generates a name automatically.

The new class command is used to define objects that belong to the class, just as **oo::object** is. By default, instances of the new class have no more behaviour than instances of **oo::object** do; new

class behaviour can be added to the class in two ways. First, a definition can be specified when creating the class; second, additional behaviour can be added to the class using **oo::define**.

The *definition*, if given, consists of a series of statements that map to the subcommands of **oo::define**. The following three code snippets are equivalent; each defines a class called *::dog* whose instances will have two subcommands: *bark* and *chase*.

```
# Method 1
oo::class dog
oo::define dog method bark {} {
    puts "Woof, woof!"
}
oo::define dog method chase thing {
    puts "Chase $thing!"
}


# Method 2
oo::class dog
oo::define dog {
    method bark {} {
        puts "Woof, woof!"
    }
    method chase thing {
        puts "Chase $thing!"
    }
}


# Method 3
oo::class dog {
    method bark {} {
        puts "Woof, woof!"
    }
    method chase thing {
        puts "Chase $thing!"
    }
}
```

4.2.2.1 Constructor

The constructor for **oo::class** concatenates its arguments and passes the resulting script to **oo::define** (along with the fully-qualified name of the created class, of course).

4.2.2.2 Methods

The instances of **oo::class** have the following methods:

create Creates a new instance of the class with the given name. All subsequent arguments are given to the class's constructor. The result of the **create** method is always the fully-qualified name of the newly-created object. Syntax:

class **create** objName ?arg arg ...?

new Creates a new instance of the class with an automatically chosen name. All subsequent arguments are given to the class's constructor. The result of the **new** method is always the fully-qualified name of the newly-created object. Syntax:

class **new** ?*arg arg ...*?

unknown Classes define an unknown-method handler. This is used to hand off attempts to create a class using the syntax:

```
oo::class foo bar
```

to the **create** or **new** method, depending on whether the class name is an empty string or not.

4.2.3 oo::definer

This metaclass (subclass of **oo::class**) arranges for its instances to have the following methods, each of which is delegated to the identically-named subcommand of the **oo::define** command described below, operating on the class instance that is an instance of **oo::definer**.

abstract, **constructor**, **destructor**, **export**, **filter**, **filterguard**, **forward**, **invariant**, **method**, **mixin**, **mixinguard**, **parameter**, **superclass**, **unexport**

Thus, the following commands are equivalent:

```
# Method 1
oo::definer dog
oo::define dog method bark {
    puts "Woof, woof!"
}

# Method 2
oo::definer dog
dog method bark {
    puts "Woof, woof!"
}
```

4.2.3.1 Constructor

The **oo::definer** constructor just passes all its arguments to its parent constructor (*i.e.*, the **oo::class** constructor).

4.2.4 oo::struct

This class (subclass of **oo::object**) has no default constructor. It has the following exported methods:

append This is the analogue of the core Tcl **append** command except that the variable name is resolved in the context of the object's namespace.

struct **append** *varName* ?*arg arg …*?

array This is the analogue of the core Tcl **array** command except that the array name is resolved in the context of the object's namespace.

struct **array** *subcommand varName* ?*arg arg …*?

eval This is a public exposure of the **eval** method defined by the **oo::object** class.

exists This is the analogue of the core Tcl **info exists** command except that the variable name is resolved in the context of the object's namespace.

struct **exists** *varName*

incr This is the analogue of the core Tcl **incr** command except that the variable name is resolved in the context of the object's namespace.

struct **incr** *varName* ?*increment*?

lappend This is the analogue of the core Tcl **lappend** command except that the variable name is resolved in the context of the object's namespace.

struct **lappend** *varName* ?*arg arg …*?

set This is the analogue of the core Tcl **set** command except that the variable name is resolved in the context of the object's namespace.

struct **set** *varName* ?*value*?

trace This is the analogue of the core Tcl **trace** command operating on variables (no other types of traceable items are supported by this method) except that variable names are resolved in the context of the object's namespace.

struct **trace** *subcommand* ?*arg arg …*?

unset This is the analogue of the core Tcl **unset** command except that the variable name is resolved in the context of the object's namespace.

struct **unset** ?*varName varName …*?

vwait This is the analogue of the core Tcl **vwait** command except that the variable name is resolved in the context of the object's namespace.

struct **vwait** *varName*

The following non-exported methods are defined:

var This method takes one argument, the name of a variable to be resolved in the context of the object's namespace, and returns the fully qualified name of the variable. This is suitable for use with core commands other

than those already supported above (*e.g.*, selected **dict** subcommands) or extensions such as Tk (*e.g.*, for the **-textvariable** option of the **label**, **button** or **entry** widgets). This method does not assign any value to the variable. Syntax:

struct **var** *varName*

It is expected that this convenience method will normally be used solely through the **my** command within the context of a method, like this:

my var *varName*

4.3 Introspection Support

The core Tcl **info** command shall be extended in the following ways:

- An **object** subcommand that shall provide information about a particular object. Its first argument shall be the name of an object to get information about, its second argument shall be a sub-subcommand indicating the type of information to retrieve and all subsequent arguments shall be arguments, as appropriate. The following types of information shall be available:

args Returns the list of arguments to a method supported by an object.

info object *object* **args** *method*

body Returns the body of a method supported by an object.

info object *object* **body** *method*

check Returns the current list of enabled assertion types for an object (see the documentation for **oo::check** for the list of acceptable assertion types).

info object *object* **check**

class Returns the class of an object, or if *className* is specified, whether the object is (directly or indirectly through inheritance or mixin) an instance of the named class.

info object *object* **class** ?*className*?

default Returns whether a particular argument to a method has a default value specified, much as **info default** does for a normal procedure argument.

info object *object* **default** *method argName defaultValueVar*

filters Returns the list of filters defined for an object.

info object *object* **filters**

filterguards Returns the list of filter-guards for a particular filter.

info object *object* **filterguards** *name*

invariants Returns the list of invariants defined for an object.

info object *object* **invariants**

isa Returns boolean information about how an object relates to the class hierarchy. Supports a range of subcommands to allow the specification of what sort of test is to be performed:

class Returns whether the named object is a class.

info object *object* **isa class**

metaclass Returns whether the named object is a class that is not of immediate type **oo::class** but rather one of its subtypes instead.

info object *object* **isa metaclass**

mixin Returns whether the named object has *mixinClassName* as one of its mixins.

info object *object* **isa mixin** *mixinClassName*

object Returns whether *object* really names an object.

info object *object* **isa object**

typeof Returns whether the object is of type *class* (*i.e.*, an instance of that class or an instance of a subclass of that class).

info object *object* **isa typeof** *class*

methods Returns the list of methods defined for an object.

info object *object* **methods**

mixins Returns the list of mixins for an object.

info object *object* **mixins**

post Returns the postcondition for the named method, or an empty string if no postcondition has been defined.

info object *object* **post**

pre Returns the precondition for the named method, or an empty string if no precondition has been defined.

info object *object* **pre**

vars Returns the list of all variables defined within the object, or optionally just those that match *pattern* according to the rules of **string match**.

info object *object* **vars** ?*pattern*?

- A **class** subcommand that shall provide information about a particular class. Its first argument shall be the name of a class to get information about, its second argument shall be a sub-subcommand indicating the type of information to retrieve and all subsequent arguments shall be arguments, as appropriate. The following types of information shall be available:

abstract Returns whether the named class is an abstract class.

info class *class* **abstract**

args Returns the list of arguments to a method supported by an object.

info class *class* **args** *method*

body Returns the body of a method supported by an object.

info class *class* **body** *method*

default Returns whether a particular argument to a method has a default value specified, much as **info default** does for a normal procedure argument.

info class *class* **default** *method argName defaultValueVar*

filters Returns the list of filters defined for an object.

info class *class* **filters**

filterguards Returns the list of filter-guards for a particular filter.

info class *class* **filterguards** *name*

instances Returns a list of all direct instances of the class (but not instances of any subclasses of the class).

info class *class* **instances**

invariants Returns the list of invariants defined for an object.

info class *class* **invariants**

methods Returns the list of methods defined for an object.

info class *class* **methods**

mixins Returns the list of mixins for an object.

info class *class* **mixins**

mixinguards Returns the list of mixin-guards for a particular mixin.

info class *class* **mixinguards** *name*

parameters Returns a list of all parameters defined by the class.

info class *class* **parameters**

post Returns the postcondition for the named method, or an empty string if no postcondition has been defined.

info class *class* **post** *method*

pre Returns the precondition for the named method, or an empty string if no precondition has been defined.

info class *class* **pre** *method*

subclasses Returns a list of all subclasses of the class, or optionally just those that match *pattern*.

info class *class* **subclasses** ?*pattern*?

superclasses Returns a list of all superclasses of the named class in the class hierarchy. The list will be ordered in inheritance-precedence order.

info class *class* **superclasses**

4.4 The oo::define Command

Syntax:

oo::define objectOrClass subcommand ?arg ...?
oo::define objectOrClass script

The **oo::define** command is used to add behaviour to objects or classes. In the second form, script is a Tcl script whose commands are the subcommands of **oo::define**; this is a notational convenience, as the two forms are semantically equivalent. (Note that the context in which *script* executes is otherwise undefined.)

4.4.1 Class-related Subcommands

The subcommands of **oo::define** (which may be unambiguously abbreviated when not in the script form) shall be:

abstract This is valid only for classes, takes no arguments, and marks the class so that instances of the class cannot be created. Subclasses may be created though; abstract-ness is not inherited.

constructor This is valid only for classes, takes two arguments (a **proc**-style argument list, and a body script), and sets the constructor for the instances of the class to be executed as defined by the body script after binding the actual arguments to the call that creates an instance of the class to the formal ar-

guments listed. The constructor is called after the object is created (following checks for abstractness) but before any instance variables are guaranteed to be set. If no constructor is specified, the constructor will accept exactly the same arguments as the constructor in the parent class, and will delegate all the arguments to that parent-class constructor. See the **method** subcommand for a description of the behaviour of pre- and postconditions.

oo::define *class* **constructor** *argList body* ?*precondition*? ?*postcondition*?

copy This creates an exact copy of an object with the given name. If *name* is omitted or the empty string, a new name will be generated automatically.

oo::define *object* **copy** ?*name*?

destructor This is valid only for classes. It defines the class destructor; a destructor is like a method but takes no arguments. It is called by the object's destroy method, which is defined automatically and which cannot be overridden. The syntax is as follows:

oo::define *class* **destructor** *body*

In classic XOTcl, the destructor is simply a method; it must explicitly call the parent destructor using XOTcl's **next** command. In **oo::** the chain of destructors is called in the proper sequence automatically and independently of the content of any particular destructor.

Note that destructors are called whenever the object is deleted by any mechanism (except when the overall interpreter is deleted, when execution of Tcl scripts has ceased to be possible anyway).

export This specifies that the named methods are exported, *i.e.*, part of the public API of the class's instances. The syntax is as follows:

oo::define *class* **export** *name* ?*name ...*?

An exported method is accessible to clients of the object; an unexported method is accessible only to the object's own code through the **my** command.

filter this subcommand (operating on the class if the object is a class, and on the object itself otherwise – see the **self.filter** subcommand for how to force it the other way) controls the list of filter methods for a class or object. Each filter method in the list is called when any method is invoked on the class's instances or the object, and it is up to the filter to decide whether to invoke the filtered method call (using the **next** command) or return a suitable replacement value.

oo::define objectOrClass **filter** filterList

filterguard This subcommand defines a list of guard expressions for a filter; the filter is skipped (*i.e.*, the underlying method call is invoked directly) if any of the guards returns a false value. Syntax:

oo::define *objectOrClass* **filterguard** *filterName guardList*

forward This subcommand (operating on the class if the object is a class, and on the object itself otherwise – see the **self.forward** subcommand for how to force it the other way) defines a class method which is automatically forwarded (*i.e.*, delegated) to some other command, according to a simple pattern. Each *arg* is used literally.

oo::define *objectOrClass* **forward** *name targetCmd* ?*arg* …?

invariant This subcommand (only valid for classes) defines a set of class invariants, scripts that must return a true value both before and after every method call. This set is inherited by subclasses. Note that invariant checking is off by default. Syntax:

oo::define *class* **invariant** *invariantList*

method This subcommand (only valid for classes) defines a class method (*i.e.*, a method supported by every instance of the class). By default, methods are exported if they start with a lower-case letter (*i.e.*, any character in `\u0061` to `\u007a` inclusive) and are not exported otherwise. The optional pre- and postconditions expressions are evaluated in the context of the body of the method; the precondition must return a true value for the method body to actually start executing, and the postcondition must return a true value after the method body has executed (unless an error was generated) for a normal method exit to happen. The default error message (on a false condition result) is "precondition failed" or "postcondition failed", but if the conditions return an error message that is used instead. If only one condition is given, it is the precondition.

oo::define *class* **method** *name args body* ?*precondition*? ?*postcondition*?

mixin This subcommand defines a mixin for a class or object, which is a way of bringing in additional method implementations (which may add to or wrap existing methods) on an *ad hoc* basis. It operates on the class if the object is a class and on the object itself otherwise – see the **self.mixin** subcommand for how to force it the other way. The list of mixins is traversed when searching for methods before the inheritance hierarchy, and mixed-in methods may chain to any methods they override using the next command.

oo::define objectOrClass **mixin** mixinList

mixinguard This subcommand defines a list of guard expressions for a mixin; the mixin is skipped (*i.e.*, the underlying method call is invoked directly) if any of the guards returns a false value. Syntax:

oo::define *objectOrClass* **mixinguard** *filterName guardList*

parameter This subcommand defines a parameter (or parameters), an instance variable with an identically named and automatically defined access method. If any name is a two-element list, the first element is the name of the variable and the second element is the default value to assign to the variable.

oo::define *class* **parameter** *name* ?*name ...*?

The access methods are always defined something like this, for a parameter named *bar* in a class named *foo*:

```
oo::define foo method bar args {
    my variable bar vbl
    if {[llength $args] == 0} {
        return $vbl
    } elseif {[llength $args] == 1} {
        return [set vbl [lindex $args 0]]
    }
    return -code error "wrong # args: ..."
}
```

superclass This specifies the superclass (or classes) of a class. Inheritance will follow the XOTcl pattern (except with a somewhat different class hierarchy, of course). Syntax:

oo::define *class* **superclass** *classList*

unexport This specifies that the named methods are unexported, *i.e.*, private. The syntax is as follows:

oo::define *class* **unexport** *name* ?*name ...*?

An exported method is accessible to clients of the object; an unexported method is accessible only to the object's own code, through the **my** command.

4.4.2 Per-Object Subcommands

The following subcommands are all per-object versions of the class subcommands listed above. When they are applied to a class, they operate on the class instance itself as an object, and not on the instances (current and future) of that class (which is why the distinction is required).

self.class This subcommand gets and sets the class of an object. Changing the class of an object can result in many methods getting added or removed.

self.export This increases the set of commands exported by the object.

self.filter This is a per-object version of the **filter** subcommand.

self.filterguard
This is a per-object version of the **filterguard** subcommand.

self.forward This is a per-object version of the **forward** subcommand.

self.invariant This is a per-object version of the **invariant** subcommand.

self.method This is a per-object version of the **method** subcommand.

self.mixin This is a per-object version of the **mixin** subcommand.

self.mixinguard
This is a per-object version of the **mixinguard** subcommand.

self.unexport This decreases the set of commands exported by the object.

4.5 Other Commands in the ::oo Namespace

4.5.1 oo::check

This controls the types of assertion checked for a particular object. The following types of assertion may be controlled:

pre When specified, states that preconditions should be checked during the processing of an object's methods.

post When specified, states that postconditions should be checked during the processing of an object's methods.

invariants When specified, states that object-defined invariants should be checked during the processing of an object's methods.

classinvariants
When specified, states that class-defined invariants should be checked during the processing of an object's methods.

The set of types of assertion to check is specified as the second argument to the **oo::check** command, the first argument being the object to set the assertion checking behaviour of. The special type **all** can be specified to select all assertion types.

oo::check *object assertTypeList*

5. XOTcl Features Omitted from the Core OO System

Object::autoname This is trivially implemented in a small procedure, and core objects can pick names for themselves and are renameable.

Object::cleanup This is not an especially well-defined method (what if the object happens to hold handles to complex resources such as network sockets; it is not

generally possible for the state of the remote server to be reset) and can be added in any compatability layer.

Object::configure This feature has been deliberately omitted from the core object system. This would be value added by any XOTcl extension.

Object::extractConfigureArg
This feature is part of configure.

Object::getExitHandler
This feature is not necessary for this version. If it existed, it would not need to be a part of the base object.

Object::info The introspection features are moved into the core **info** command.

Object::move This feature is equivalent to the use of the standard **rename** operation.

Object::noinit This feature has been deliberately omitted from the core object system because its use is dependent on the use of other deliberately-omitted features (*i.e.*, **Object::configure**). This would be value added by any XOTcl extension.

Object::parametercmd
The core object system always handles parameters in the same simple way; customisation of this process should be done by subclasses of **oo::class** that override the parameter method.

Object::requireNamespace
It should be possible to do away with this feature through better integration with the core.

Object::setExitHandler
See the comments for **Object::getExitHandler** above.

Class::__unknown Auto-loading of unknown classes is handled by the standard core unknown command.

Class::allinstances This feature is trivially implemented in a small procedure.

Class::alloc The core objects have no default behaviour, so the difference with the basic core class behaviour is moot.

Class::create Core object creation is a much more sealed process, but the lack of configure-like behaviour means that the complexity of this method is not necessary. Instead, constructors are called automatically.

Class::parameterclass
Core object system parameters are not implemented by classes.

Class::volatile This feature is omitted.

6. Suggested Class Hierarchy for XOTcl Support

The XOTcl *Object* class should derive from the core **oo::struct** class. The XOTcl *Class* class must derive from the core **oo::definer** and the XOTcl *Object* classes. This gives the following diagram (core classes are yellow boxes with their labels in lower case and with their namespace omitted, XOTcl classes are blue boxes with capitalized labels).

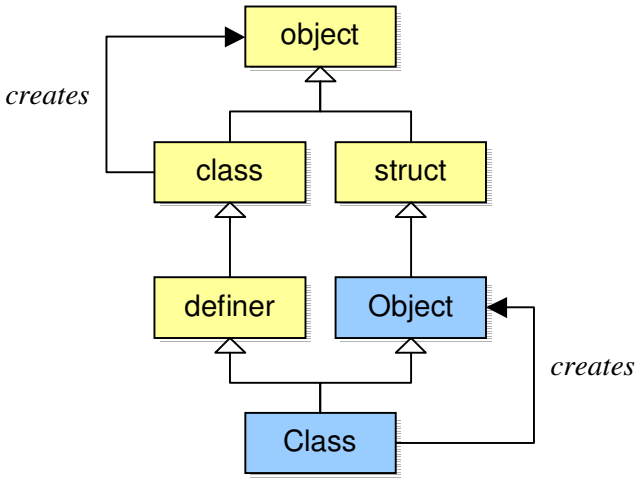

Figure 1: Base Class Hierarchy for XOTcl Clases

Note that **class** instances create **object**s (or subclasses thereof), but *Class* instances create *Object*s (or subclasses thereof).

7. References

[incr Tcl] http://incrtcl.sourceforge.net/itcl/
OTcl http://bmrc.berkeley.edu/research/cmt/cmtdoc/otcl/
Snit http://www.wjduquette.com/snit/
stooop http://jfontain.free.fr/stooop.html
tcllib, http://tcllib.sourceforge.net/
XOTcl http://media.wu-wien.ac.at/