# An Overview of the Next Scripting Toolkit

**Gustaf Neumann and Stefan Sobernig**
Tcl/Tk 2011 Conference, October 2011

WU Vienna, University of Economics and Business
Vienna, Austria

## Topics of this Presentation

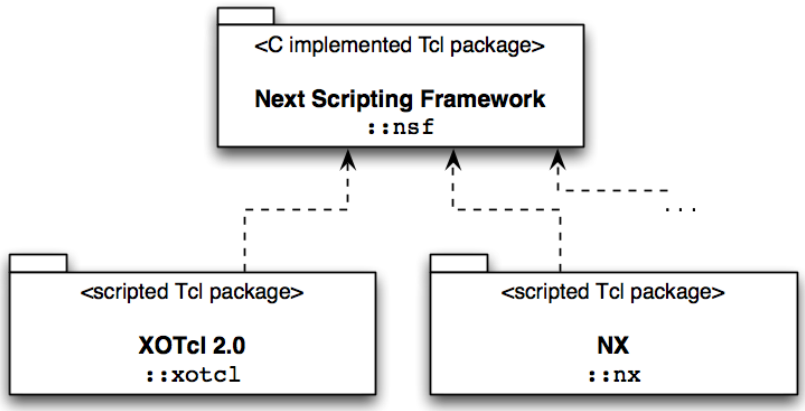
- Basic ideas of the Next Scripting Framework (NSF)
- The support for Language-Oriented Programming (LOP)
  - Scripting object systems
  - Behavioral feature composition: Method aliasing, traits
- The design goals of the Next Scripting Language (NX)
  - Robust object construction: Object parameters and initialization scripts
  - Hierarchical method interfaces: Method ensembles
  - Module encapsulation: Method call protection
- Performance, features, status, availability

## Motivation

- **Programming in the large:**
  - Mio LOCs, varying programmer teams
  - Millions of objects, hundred thousands of classes
  - Running in a seriously multithreaded environment (e.g. on average 30 threads)
  - Several thousand concurrent users
  - "Web scaled" systems
  - Today: up to 200 view per seconds, every view 10+ SQL queries, 5 mio page views/day
- **Focus on dynamically evolvable systems:**
  - Dyamnically extend systems without reboot
  - Increase number of participants in design, somewhat similar to "Wisdom of the crowds"
  - Instruments: Domain Specific Languages, reflective systems, interceptors

## Next Scripting Framework (NSF)

- The name is derived from the universal method combinator "next", as introduced by XOTcl.
- Framework: implemented in C, loadable as a Tcl package `::nsf`
- Object systems are fully scripted, organized as Tcl packages: e.g. ::xotcl, ::nx
- NSF can host multiple object systems in a single Tcl interpreter



## Scripted Object Systems

The Next Scripting Framework is the toolbox for the object-system implementor:

```
# Create an object system with the base classes named "myObject" and "myClass"
nsf::objectsystem::create myObject myClass

# The base classes are created with no methods defined.
#
# Provide methods to create and delete objects/classes based on
# predefined methods set using arbitrary names

nsf::method::alias myClass  + nsf::methods::class::create
nsf::method::alias myObject - nsf::methods::object::destroy

# Create an application class and an instance using the method "+":
myClass + C
C + c1

# Delete the instance using the method "-":
c1 -
```

Next steps:

- Define selected relations between the core meta-objects (e.g., meta-class/class, class/instance, mixin relationships )
- Populating object system with behavior offered in a Tcl- and Tcl/C-implemented method pool.

## Design Goals of NX

- The Next Scripting Language (NX) is a descendant of XOTcl
- The design goals of NX shaped the feature set of NSF
- Many NX features are available for XOTcl 2 as well
- NX differs from XOTcl in the following respects:
  1. **Stronger Encapsulation:**
     - Goal: Easing reuse, encouraging the definition of explicit interfaces
     - By default, NX does not provide any general instance variable accessor method (`set`), nor any universal variable importer (`instvar`).
     - Instead: NX provides access to instance variables via *variable resolvers*
     - Method redefinition protection, method call protection
     - Approach: Encapsulation by convenience
  2. **Framework supporting Feature Composition**:
     - Method aliasing, traits
     - Extensible method ensembles

## Design Goals of NX (2)

3. **Orthogonal Parameterization:**
   - Same parametrization mechanisms for methods and objects
   - A single argument parser is used for
     - Scripted Methods
     - C-implemented methods and Tcl commands
     - Object Parametrization
   - Highly orthogonal parameter value checking
   - Orthogonal introspection: same interface to query (introspect) C-implemented and scripted methods/commands.
4. **Robust Object Initialization**
   - Integrated object parametrization
   - Tclish scripted init-blocks instead of the vararg dash-commands in XOTcl

## Design Goals of NX (3)

5. **Flattening the Learning Curve:**
   - Introducing mainstream naming conventions
   - Smaller naming footprint: The number of methods in NX is approx. a third of the interface size in XOTcl
6. **Better Debugging and Profiling Support**:
   - Integrated profiler
   - DTrace support

## Method Aliasing

- Method Aliasing
  - ... is a composition technique to
    - bind *freestanding*, *shareable* method implementations (Tcl procs, Tcl/C commands, methods, objects)
    - ... as *method members* to arbitrary objects/classes,
    - ... via an *arbitrary name*.
  - ... realizes an important kind of *behavioral implementation reuse* (e.g. when specifying the behavior of an object system).
- Method aliases
  - ... are *transparent for method introspection*, i.e., the method parameter specification of the alias target are returned.
  - ... provide the infrastructure for realizing
    - ... support for advanced feature composition, e.g., *traits*.
    - ... *method ensembles*

## Example for Method Aliases in NX

- Method aliases and method forwarders can bind C-implemented or scripted implementations to methods (including arbitrary Tcl commands)
- Methods aliases can make use of certain features of NSF such as method protection or return value checking for these implementations

```
nx::Class create C {
  :property {a 0}

  # scripted method
  :public method add {x:integer y:integer} -returns integer {
    return [expr {$x + $y}]
  }

  # Method alias and method forward
  :public alias incr -returns integer -frame object ::incr
  :public forward plusOne -returns integer ::expr 1 +
}

C create c1     ;# create instance c1
c1 incr a       ;# increments instance variable "a" to 1
c1 incr a       ;# increments instance variable "a" to 2

puts [c1 a]     ;# outputs 2
puts [c1 plusOne [c1 a] * 100] ;# outputs 201
puts [c1 a]     ;# outputs 2
```

## Support for Advanced Feature Composability: Traits

- NX supports per-object, and transitive per-class mixins (like XOTcl)
- Traits provide an alternative *composition mechanism* for method reuse which overcomes the "total composition ordering" limitation of mixins
- Traits are like "partial classes" *defining* required missing pieces
- Traits provide more *fine-grained control* over composition
- Traits can be simple or complex (nested traits)

## Robust Object Initialization

- Objective: Robust object construction by protecting against
  - *constructor anomalies* (e.g., method shadowing, interruption of `next` chains), and
  - argument parsing ambiguities (dash-methods in XOTcl)
- NX constructs an object in four steps:
  a. *Creation:* Providing for object storage
  b. *Parametrization:* using a flattened object parameter specification (including defaults)
  c. *Init-block;* One-time evaluation of a nested script for the scope of the constructed object.
  d. *Constructor:* The `init` method chain is processed
- The object parameter specification is computed from the object properties and the object variables defined.
- Object parameters leverage the parameter type checkers available (e.g., `string is` value classes).

# A Constructor Anomaly

Consider an example of a constructor anomaly, as found in literature, coded in XOTcl:

1. Define a base class `A`:

```
xotcl::Class create A
A instproc init args {
  my m
}
A instproc m {} {
  # ...
}
```

2. Define a subclass `B`, possibly defined by a different Tcl package:

```
xotcl::Class create B -superclass A -parameter {b}
B instproc init {s} {
  next    ; # dispatching A.init()
  my instvar b
  set b $s
}
B instproc m {} {
  my instvar b
  return $b
}

B create b1 "ZAP!"; # --> can't read "b": no such variable while executing "return $b"
```

*Issue*: Method `m` is called from constructor of class `A` before the instance variable `b` is set, breaking the assumption in method `m` of class `B`.

# Avoiding the Constructor Anomaly in NX

- Use required object parameters (*properties*) instead of passing arguments into the constructor

```
nx::Class create A {
  :method init {} {
    :m
  }
  :public method m {} {
    # ...
  }
}

nx::Class create B -superclass A {
  :property b:required
  :public method m {} {
    return ${:b}
  }
}

B create b1 -b "ZAP!"
B create b2; # --> required argument 'b' is missing, should be: ::b2 configure -b ...
```

## Method Ensembles

- Resemble Tcl's idiom of sub-commands and namespace ensembles
- Objective: Bind heterogeneous methods (i.e., having distinct signatures) under common, composite method names
- Avoids the need for complex conditional branching, e.g., extensive switch threading, and custom argument parsing
- Benefits:
  - Supported by *method introspection*
  - Integrated with standard *unknown handling*
  - Accessible to *method combination* (i.e., mixins and filters)
  - Method ensembles are *extensible* via subclasses, mixins etc.
- Implemented as a special kind of object delegation hierarchies, using method aliases

## Method Ensembles without Language Support

Define a method `foo` with sub-methods `sub1` and `sub2` and an unknown handler without language support (e.g., in XOTcl):

```
Object create o
o instproc foo {sub args} {
    #
    # Define sub-methods behavior via "switch" statement
    #
    switch -exact -- $sub {
      sub1 {
        # ensemble method 'foo sub1': provide a custom parser for "args"
      }
      sub2 {
        # ensemble method 'foo sub2': provide a custom parser for "args"
      }
      default {
        # unknown handling
        set m "[current method]: unknown sub-method '$sub'. Available: sub1 sub2"
        return -code error $m
      }
    }
  }
}

o foo sub1 arg1 arg2; # OK
o foo sub2 -np1 arg1 -np2 arg2 arg3; # OK
o foo sub3; # --> foo: unknown ensemble method 'sub3'. Available: sub1 sub2
```

## Method Ensembles with Language Support

Example in NX:

```
Object create o {
  :public method "foo sub1" {p1 p2} {
    # ...
  }
  :public method "foo sub2" {-np1 -np2 p3} {
    # ...
  }
}

o foo sub1 arg1 arg2; # OK
o foo sub2 -np1 arg1 -np2 arg2 arg3; # OK
o foo sub3;
# --> Unable to dispatch sub-method "sub3" of ::o foo;
#     valid are: foo sub1, foo sub2
```

## Method Call Protection

- NX supports stronger means for module encapsulation than XOTcl: no default setters/getters (e.g., XOTcl's `set` or `instvar`), redefinition protection, and *call protection*.
- An NX object can expose three kinds of callable methods:
  - *public*: Methods callable from any client scope (i.e., self-, command- , and next-calls).
  - *protected*: Methods available only for self- and next-calls.
  - *private*: Methods available for self-calls issued from within the same class or object as the call target.
- Private methods are important for e.g. feature composition to avoid unexpected shadowing
- Method call and redefine protection provided by NSF method-properties

## Method Call Protection in NX

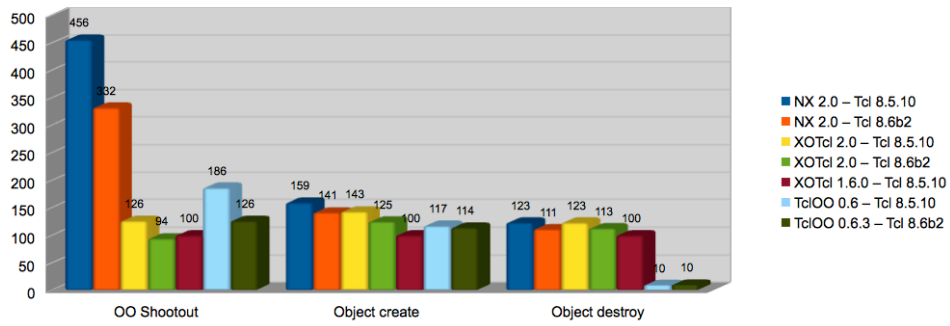
```
nx::Class create A {
  #
  # Public interface of class "A"
  #
  :public method foo args {
    :bar              ; # invoke protected method of current object
  }

  #
  # Protected interface of class "A"
  #
  :protected method bar {} {
    : -local baz      ; # invoke private method of current object with "-local" flag
    # ...
  }

  #
  # Private interface of class "A"
  #
  :private method baz {} {
    # ...
  }
}
A create ::a1
```

## Performance Improvements

Performance improvements relative to XOTcl 1.6.0 (index = 100):

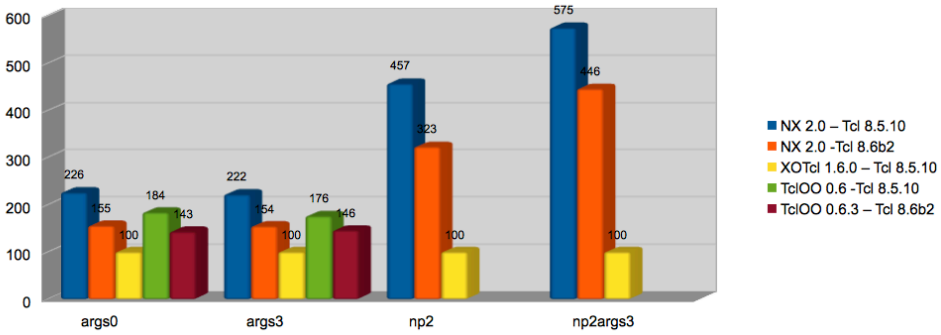



Summary:

- NX is quite fast although it is scripted
- On some tests, NX is nearly 5 times faster than XOTcl 1.6.0
- Both, NX and TclOO perform better on Tcl 8.5.10 than on Tcl 8.6b2

## Performance Improvements (2)

Performance improvements on method dispatches (compared to XOTcl 1.6.0)




Summary:

- Same overall picture as for last slide
- On some tests, NX is more than 5 times faster than XOTcl 1.6.0

## Summary

The major contributions of NSF/NX are:

- Improved encapsulation
- Improved feature composition
- Easier and earlier error detection
- Enhancements to the concrete syntax
  - Parametric objects, method ensembles
  - Parameter types, parameter options
- Improved performance and scalability
- Improved Tool Support
  - DTrace integration
  - Tcl/C-API generator
  - Various: * functional testing with `nx::test`, documentation generation with `nxdoc`, profiling, MongoDB/NX binding, ...

Available from http://next-scripting.org/