XOTcl @ Work

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

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

What is XOTcl

◆ XOTcl = Extended Object Tcl
◆ “High-level” object-oriented programming
◆ Advanced Component Glueing

◆ XOTcl is freely available from: http://www.xotcl.org

◆ Outline:
  – Scripting and object-orientation
  – Programming the “basic” XOTcl Language
  – Component Glueing
  – XOTcl high-level language constructs
  – Some provided packages
Tcl-Strengths

Important Ideas in Tcl:

◆ **Fast & high-quality development through component-based approach**

◆ **2 levels: “System Language” and “Glue Language”**

◆ **Flexibility through . . .**
  – Dynamic extensibility,
  – Read/write introspection,
  – Automatic type conversion.

◆ **Component-Interface through Tcl-Commands**

◆ **Scripting language for glueing**
Motivation for XOTcl

◆ Extend the Tcl-Ideas to the OO-level.

◆ Just “glueing” is not enough! Goals are . . .
   – Architectural support
   – Support for design patterns (e.g. adaptations, observers, facades, . . .)
   – Support for composition (and decomposition)

◆ Provide flexibility rather than protection:
   – Introspection for all OO concepts
   – All object-class and class-class relationships are dynamically changeable
   – Structural (de)-composition through Dynamic Aggregation
   – Language support for high-level constructs through powerful interceptors
     (Filters and Per-Object Mixins)
XOTcl Overview

**Tcl**
- namespaces
- introspection
- extensibility
- embeddability
- dynamic type system with automatic conversion
- language dynamics

**Extended OTcl**

*New Functionalities:*
- dynamic aggregations
- nested classes
- assertions
- per-object mixins
- per-class mixins
- filters
- scripted components

*Adopted from OTcl:*
- object and class system
- multiple inheritance
- method chaining
- meta-classes
- read/write introspection
- dynamic typing

*Other Extensions*
XOTcl is similar Tcl

◆ XOTcl is dynamic:
  – Definitions of objects and classes can be extended and modified at runtime
  – Classes and objects can be dynamically destroyed
  – All relationships between object and classes are fully dynamic

◆ XOTcl is fully introspectible with info methods

◆ Syntax similar to Tcl

◆ Objects and classes are Tcl commands

◆ Objects and classes “live” in a Tcl namespace
Example: Soccer Team

- Schalke
- Gelsenkirchen

Operations:
- new player
- transfer player

Emile Mpenza
- name
- role Forward

◆ Soccer team abstraction:
- Has members (players)
- Has properties (name, location, type)
- Players can be added and transferred
- Each player has properties (name, player role)

◆ Similar abstractions in many “real-world” applications
Soccer Team In Ordinary Tcl

set teams($teamid-name) "Schalke" ;# Associative array for teams
set teams($teamid-location) "Gelsenkirchen"
set teams($teamid-playerids) {}

set $id-players($playerid-name) "Emile Mpenza" ;# Player array for each team
...

proc newPlayer {teamid name} {
    global teams $teamid-players
    ...
    return $playerid
}

Problems: Missing data encapsulation, global data, name collision, no bundled behavior/data, no specialization/generalization, central modification is hard to achieve,
...
Object-Oriented Solution

◆ Initial Design: Soccer team aggregates players.

◆ Used Concepts:
  – Classes abstract over soccer team and player
  – Instance variables
  – Instance methods
  – 1-to-many relationship
  – (Dynamic) object aggregation
Objects in XOTcl

◆ Each created object has `Object` as class or superclass. Methods on `Object` are usable for all objects.

◆ Each object can have object-specific variable slots and methods (`procs`).

◆ Variables and methods are stored in the object’s namespace.

◆ Each object has a class.
Creation and Definition of Objects

Object player1

player1 set name "Emile Mpenza"

player1 proc print {} {
    [self] instvar name
    puts "Name: $name"
}

player1 print

player1 destroy
Objects versus Classes

◆ Instances (objects) can be derived from a class

◆ A class describes the intrinsic type of an object:
  – Common data slots
  – Instance methods (instprocs)
  – ...

◆ Classes in XOTcl “know” about their instances and vice versa (introspection)

◆ Classes in XOTcl have all object abilities plus class abilities:
  – Deriving objects
  – Instance method definition
  – Inheritance
  – ...
Class Instances

Classes

Objects

player1
player2
player3
player4
team1
team2

is-instance-of relationship

instance-of

Gustaf Neumann, Uwe Zdun
University of Essen
Class Definition and Instance Methods on Classes

Class Player -parameter {
    name
    {playerRole NONE}
}

Player instproc print {} {
    [self] instvar name playerRole
    puts "Name: $name"
    puts "Player Role: $playerRole"
}

Player emile -name "Emile Mpenza" -playerRole Forward

emile print

Stepwise refinement of class definition, syntax & conventions similar to Tcl
Object Construction/Destruction

◆ Constructor – Special instance method `init`:

```tcl
Player instproc init args {
    # perform initializations
}
Player p -name "My Name"
```

◆ Destructor – Special instance method `destroy`:

```tcl
Player instproc destroy args {
    # perform destruction
}
p destroy
```
Introspection

◆ In XOTcl every language is introspective and dynamic ⇒ Similar to Tcl.

◆ Using the info instance method.

◆ Example – Reading instproc definition:
  
  Player info instbody print

◆ Example – List of instances:

  Player info instances

◆ Object- vs. class-specific introspection options. Example – Obtaining an object’s class:

  player1 info class
Callstack Information

◆ Retrieve information that is dynamically created on the callstack:

<table>
<thead>
<tr>
<th>self</th>
<th>current object name</th>
</tr>
</thead>
<tbody>
<tr>
<td>self class</td>
<td>current class name</td>
</tr>
<tr>
<td>self proc</td>
<td>current proc/instproc name</td>
</tr>
<tr>
<td>self callingobject</td>
<td>calling class name</td>
</tr>
<tr>
<td>self callingclass</td>
<td>calling object name</td>
</tr>
<tr>
<td>self callingproc</td>
<td>calling proc/instproc name</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

◆ Example – Discriminating on calling object type:

Player instproc reactOnPlayer {} {
  set co [self callingobject]
  if {[co istype Player]} {...} # type => player-specific behavior
  ...
} # else: default behavior

# example instproc
# get calling object
Inheritance

- Defining a class hierarchy with “is-a” relationships
- Generalization/specialization ⇒ Reusing class definitions

```
Class ClubMember -parameter {name}
Class Player -superclass ClubMember -parameter {{playerRole NONE}}
Class Trainer -superclass ClubMember
Class President -superclass ClubMember
```
Multiple Inheritance

- Multiple Inheritance = one class has more than one superclass

- Directed Acyclic Graph

→ Linearization with Method Chaining

Class PlayerTrainer -superclass {Player Trainer}
Method Overloading and Next Path

◆ Each method call is performed on an object

◆ If the method is not defined on the object, then the class and its superclasses are searched

◆ If the method is found it may contain a next call.

◆ Then the “next” method on the class graph is searched and mixed into the current method

◆ “next” determines if, at which position, and with which arguments the next method is called

◆ Per default, “next” calls with the same arguments
Method Chaining: Extending Print Operation

Class ClubMember -parameter {name} ;# Class definition
ClubMember instproc print {} { ;# Default print operation
    [self] instvar name
    puts "Name: $name"
    next
}
Class Player -superclass ClubMember \ ;# Subclass definition
    -parameter {{playerRole NONE}}
Player instproc print {} { ;# Extended print operation
    [self] instvar playerRole
    puts "Player Role: $playerRole"
    next
}

Composability: next functions without naming the targeted superclass.
Method Chaining: Next Path for Player Trainer

Class-Path Linearization: Each class is visited once. Unambiguous precedence order.
XOTcl Class and Object System

- Common Root Class
- Object
- Class
- Meta-Classes
- Classes
- Objects
Dynamic Re-Classing

- **Dynamic classes and superclasses** ⇒ **Modeling life-cycle of objects.**

- **Example** – **Player becomes president:**

  ```tcl
  Player p -name "Franz Beckenbauer" \
  -playerRole PLAYER
  ...
  $fb class President
  ;# Create player
  ;# Life-cycle induces change
  ;# Reclassing to President
  ```

- **Redefining class behavior may imply modifications** → **specializing** class:

  ```tcl
  Player instproc class args {
      [self] unset playerRole
      next
  }
  ;# Specializing class operation
  ;# Delete player role property
  ;# Call Object->class
  ```
Dynamic Object Aggregation

*Dynamic object aggregation:* An object system supports dynamic aggregation iff arbitrary objects may be aggregated or disaggregated at arbitrary times during execution.

```
Class Stadium
Class SoccerTeam
SoccerTeam instproc init args {  
  Stadium [self]::homeStadium
  next
}
SoccerTeam bayern
President bayern::president \
  -name "Franz Beckenbauer"
bayern::president destroy
```

;# Class for stadium
;# Soccer team class
;# Constructor
;# Automatically aggregate stadium
;# New team instantiation
;# Aggregate president
;# President leaves club -> disaggregate
**Object Aggregation – Examples**

Aggregate with autoname:

```tcl
SoccerTeam instproc newPlayer args {
    eval Player [self]:::[[self] autoname player%02d] $args
}
```

Iterate over children:

```tcl
SoccerTeam instproc printMembers {} {
    puts "Members of [[self] name]:"
    foreach m [[self] info children] {puts "$m name"}
}
```

Retrieving club name from parent:

```tcl
ClubMember instproc getClubName {} {
    return [[[self] info parent] name]
}
```
Object Aggregation – Life-Cycle Issues

◆ **Object creation**: Every object is created with an identifier that is unique in the scope where it was created.

◆ **Object hierarchy restructuring**: A copy/move/delete operation works on the subtree of the object hierarchy starting with the named object.

```tcl
SoccerTeam instproc transferPlayer {playername destinationTeam} {
    foreach player [[self] info children] {
        if {([$player istype Player] && [$player name] == $playername)} {
            $player move [set destinationTeam]::[$destinationTeam autoname player%02d]
        }
    }
}
```

◆ **Object aggregation implies that the whole has responsibility of the life-time of the parts**
Dynamic Component Loading in XOTcl

◆ **Component in XOTcl:**

  – Any assembly of several structures, like objects, classes, procedures, functions, etc.
  – Granularity: self-contained entity, i.e. subsystem or substantial part of a subsystem

◆ **Component has to declare its name and optional version information with:**

  ```package provide componentName ?version?```

◆ **Component can be loaded with:**

  ```package require componentName ?version?```

◆ **Automatic component indexing, tracking, and tracing**
Component Wrapping

Component Wrapper: White-box placeholder for (multi-paradigm) components → Place for central adaptations, decorations, etc.
Wrapping a C Component with Explicit Export/Import

- Many different component wrapping schemes: Wrapper Facade, Proxy, . . .
- Different configurations: Tcl C Wrapper, XOTcl C Wrapper, . . .
- Three-Level Component Configuration: Make export and import explicit, first-class objects
  → Dynamic, runtime replaceability
Problems of a Pure Class-Based Implementation

◆ **Transparency** – The client should not rely on concrete implementation details.

◆ **Decoration/Adaptation:**
  – Concerns that cross-cut the component wrapper hierarchy,
  – Object-specific component wrapper extensions or adaptations.

◆ **Coupling of Component and Wrapper**
  – Should appear as one runtime entity,
  – But: Should be decomposed in the implementation.

◆ **Component Loading** – Dynamical and Traceable

⇒ **Interception Techniques for Flexible Component Wrapping**
Per-Object Mixins for Object-Specific Extensions

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.

- Model behavioral extension for individual objects (Decorator).
- Model Adapter for individual objects.
- Handle orthogonal aspects not only through multiple inheritance.
- Intrinsic vs. extrinsic behavior, similar to roles.
Example Code for Per-Object Mixins

Player bayern::franz \  
   -name "Franz Beckenbauer"  ;# Player object

Class Singer  
Singer instproc sing text {  ;# Define the singer class
   puts "[[self] name] sings: $text, lala."  ;# Singing method
}

bayern::franz mixin Singer  ;# Register class as per-object mixin

bayern::franz sing "lali"  ;# Perform singing

bayern::franz mixin {}  ;# Better stop it
**Per-Class Mixins**

A per-class mixin is a class which is mixed into the precedence order of the instances of a class and all its subclasses.

*Example* – Observing the player transfer operation:

```tcl
Class TransferObserver

TransferObserver instproc transferPlayer \{pname team\} {
  puts "Player '$pname' is transferred."
  puts "Destination Team '$team'"
  [self] set transfers($pname) $team
  next
}

SoccerTeam instmixin TransferObserver

bayernMunich transferPlayer \"Giovanne Elber\" chelsea
```

Gustaf Neumann, Uwe Zdun
University of Essen
Architectural Constraints

◆ **Restrict dynamic classes of sub-hierarchy to be static.**

◆ **Requests are split objects with C++ objects ⇒ Dynamic classing is impossible.**

```tcl
Class RestrictToSubClassOfRequest
RestrictToSubClassOfRequest instproc class args {
    set cl [[self] info class]
    next
    if {![self] istype Request]} {
        [self] class $cl
    }
}
Request instmixin RestrictToSubClassOfRequest
```
Filters for Cross-Cutting Concerns

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

→ Aspects that cross-cut several classes in a hierarchy.
Example: Trace Filter Definition

package provide xotcl::Traced 0.8

...  

Class Traced -superclass Class

Traced instproc traceFilter args {
    set r [self calledproc]
    if {[[[self regclass] exists operations($r)]} {
        puts stderr "CALL [self]->$r"
    }
    return [next]
}

Traced instproc init args {
    [self] array set operations {}  
    next
    [self] filterappend Traced::compositeFilter  
}

Gustaf Neumann, Uwe Zdun
University of Essen
Example: Traced Filter Usage

```tcl
package require xotcl::Traced

... # Load component dynamically

Traced ClubMember \
-AddOperations {name ...}

Class Player -superclass ClubMember

Class President -superclass ClubMember

# Define traced class
# Add traced operations

# Define different subclasses
# => They are also traced now
```
Self-Documentation

◆ XOTcl contains self-documentation/metadata facility with @

◆ Components:
  – Static metadata analysis,
  – Dynamic metadata analysis,
  – HTML generation.

◆ Syntax similar to definition of described constructs.

◆ Flexibly extensible with new tokens and properties.

◆ Per-default: not interpreted ⇒ no memory/performance wasted, if runtime metadata is not required.
Self-Documentation Examples

◆ Example – Describing a class:

@ Class SoccerTeam {
    description {A soccer team class.}
}

◆ Example – Describing a method:

@ SoccerTeam instproc transferPlayer {
    player "name of the player to transfer"
    team "destination team"
} {
    Description {
        Move player object into destination team.
    }
    return "empty string"
}
XOTcl Component Library & Application

◆ XOTcl contains rich component library:
  – Object persistence
  – XML parser and interpreter framework
  – RDF parser and interpreter framework
  – HTTP Server
  – Client-side of various web protocols (HTTP, FTP, LDAP, ...)
  – ActiWeb: Active Web Objects and Mobile Code
  – Reusable pattern implementations
  – ...

◆ Example Applications

◆ Regression Test Suite

◆ Documentation (Tutorial, Language Reference, Papers, Articles, etc.)
How to get involved

◆ Download XOTcl from http://www.xotcl.org
  – Source Distribution
  – Linux Binaries & RPMs (Red Hat, Debian)
  – Windows Binaries

◆ Mailing List with Archives at:
  http://wi.wu-wien.ac.at/mailman/listinfo/xotcl