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

Second European Tcl/Tk User Meeting, June, 2001.

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

Gustaf Neumann, Uwe Zdun
University of Essen

Tcl-Strengths

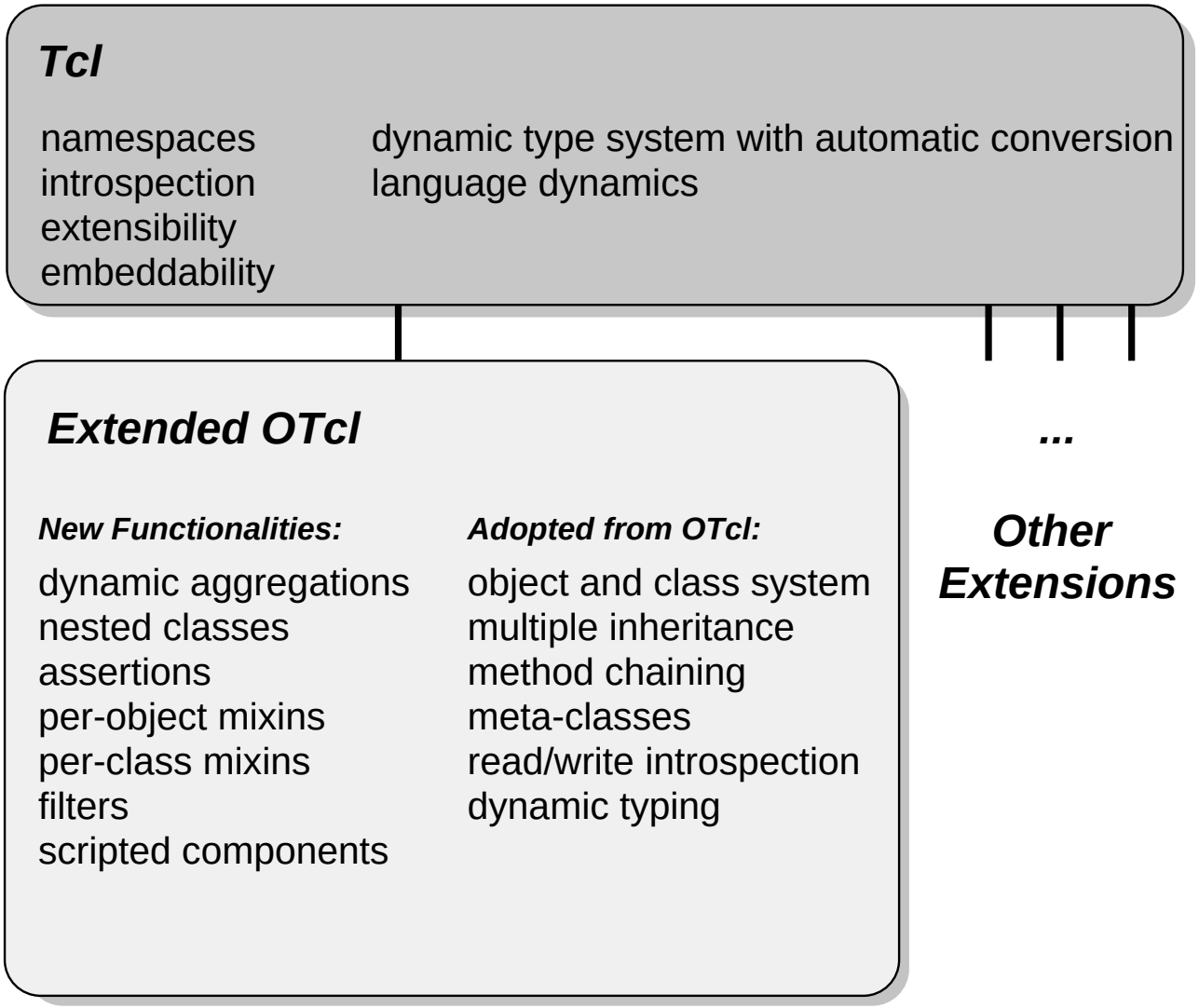
Important Ideas in Tcl:

- **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*)

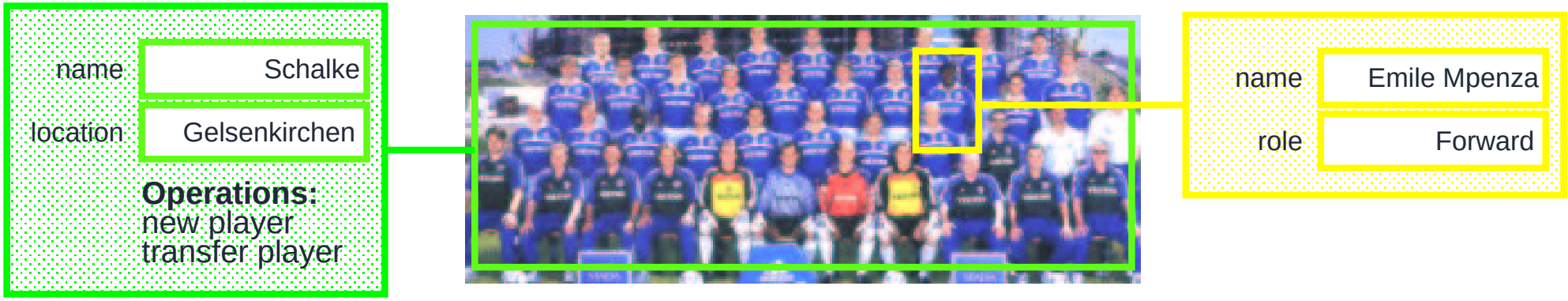
XOTcl Overview



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



- **Soccer team abstraction:**
  - Has members (players)
  - Has properties (name, location, type)
  - Players can be added and transfered
  - Each player has properties (name, player role)
- **Similar abstractions in many "real-world" applications**

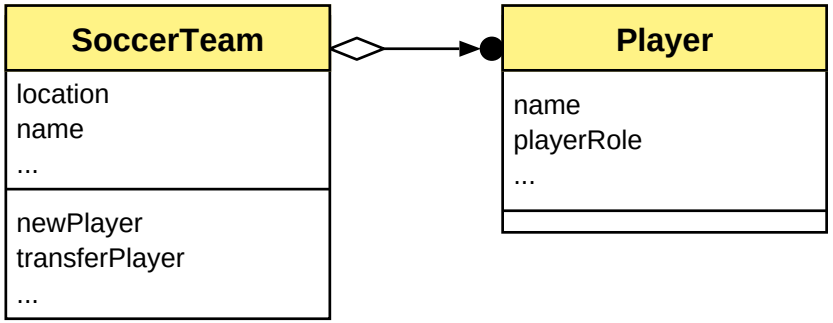
Soccer Team In Ordinary Tcl

```
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"   ;# associative array for each team
...

proc newPlayer {teamid name} {                   ;# procedure
  global teams $teamid-players                   ;# import global structure
  ...                                            ;# work on global structure
  return $playerid
}
```

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

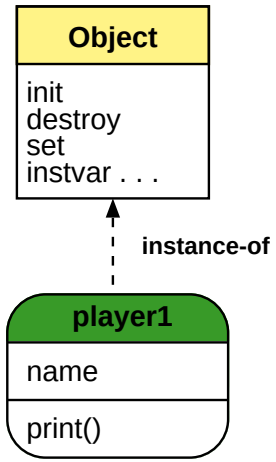
Object-Oriented Solution

- **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.**



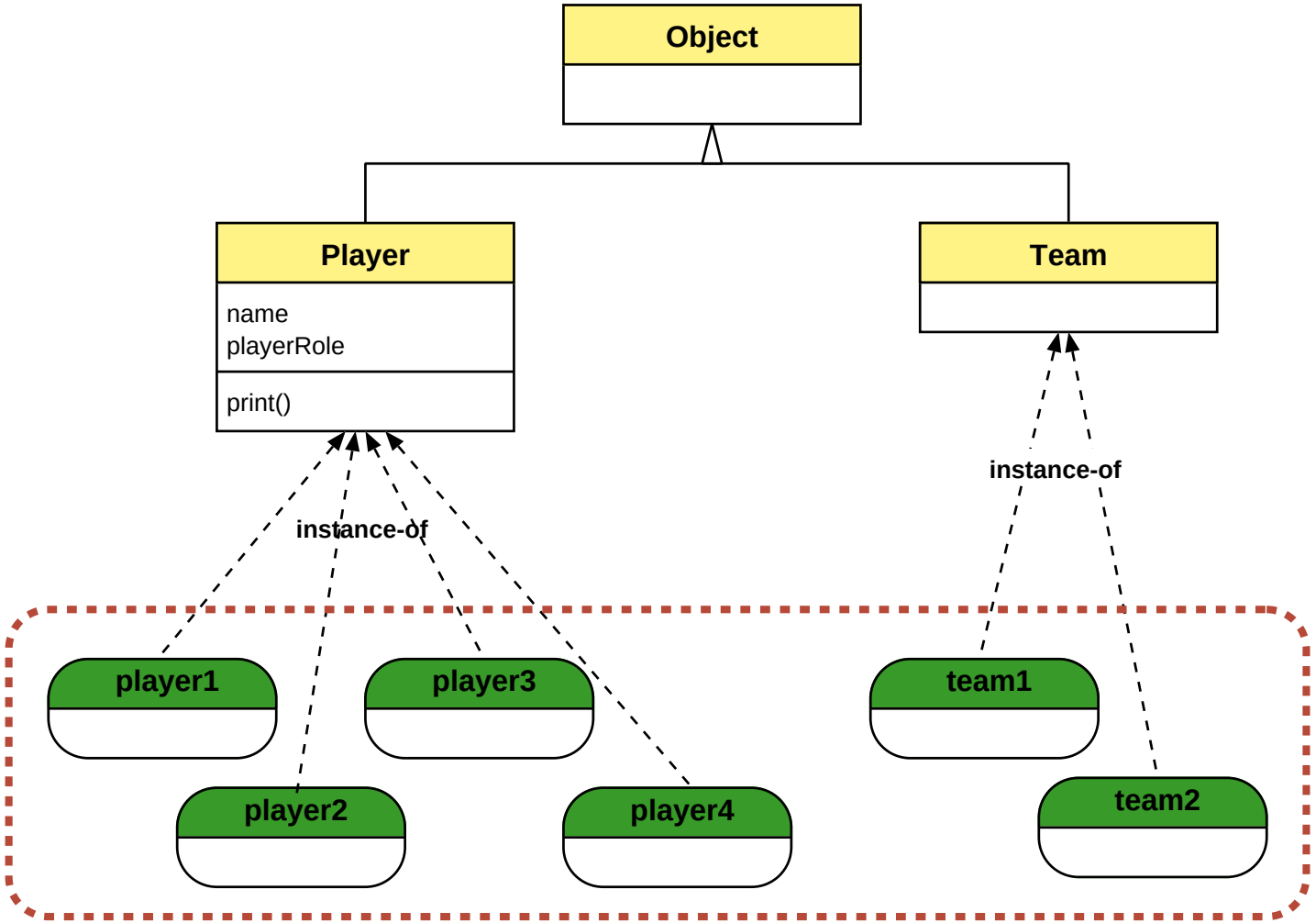
Creation and Definition of Objects

```
Object player1                                  ;# Object definition

player1 set name "Emile Mpenza"                 ;# Set instance variable

player1 proc print {} {                         ;# print procedure for name
  [self] instvar name                           ;# get instance variable into proc sc
  puts "Name:        $name"                     ;# print name to stdout
}

player1 print                                   ;# call ‘‘print’’

player1 destroy                                 ;# and delete player object
```

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

is-instance-of relationship

Objects

Class Definition and Instance Methods on Classes

```
Class Player -parameter {                          ;# Class definition
  name
  {playerRole NONE}
}
Player instproc print {} {                         ;# Print instance method
  [self] instvar name playerRole
  puts "Name:       $name"
  puts "Player Role: $playerRole"
}

Player emile -name "Emile Mpenza" \                ;# Definition of a player object
  -playerRole Forward

emile print                                        ;# Calling print operation
```

Stepwise refinement of class definition, syntax & conventions similar to Tcl

Object Construction/Destruction

- **Constructor: Special instance method `init`:**

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

- **Destructor: Special instance method `destroy`:**

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

Using Objects

- **Setting the name of an object:** `player set name "Paul Breitner"`
- **Add player by calling a method :**

```
bayernMunich newPlayer -name "Franz Beckenbauer"  -playerRole PLAYER
```

Introspection

- **In XOTcl every language is introspective and dynamic $\Rightarrow$ 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:**

`self`	current object name
`self class`	current class name
`self proc`	current proc/instproc name
`self callingobject`	calling class name
`self callingclass`	calling object name
`self callingproc`	calling proc/instproc name
...	...

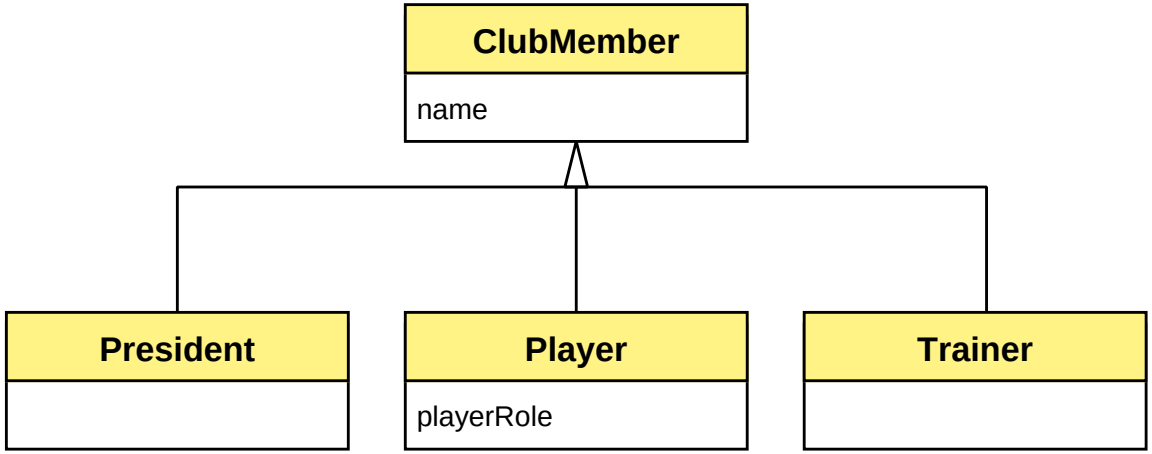
- **Example – Discriminating on calling object type:**

```
Player instproc reactOnPlayer {} {
  set co [[self] info callingobject]
  if {[$co istype Player]} {...}
  ...
}
```

```
# example instproc
# get calling object
# type => player-specific behavior
# else: default behavior
```

Inheritance

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

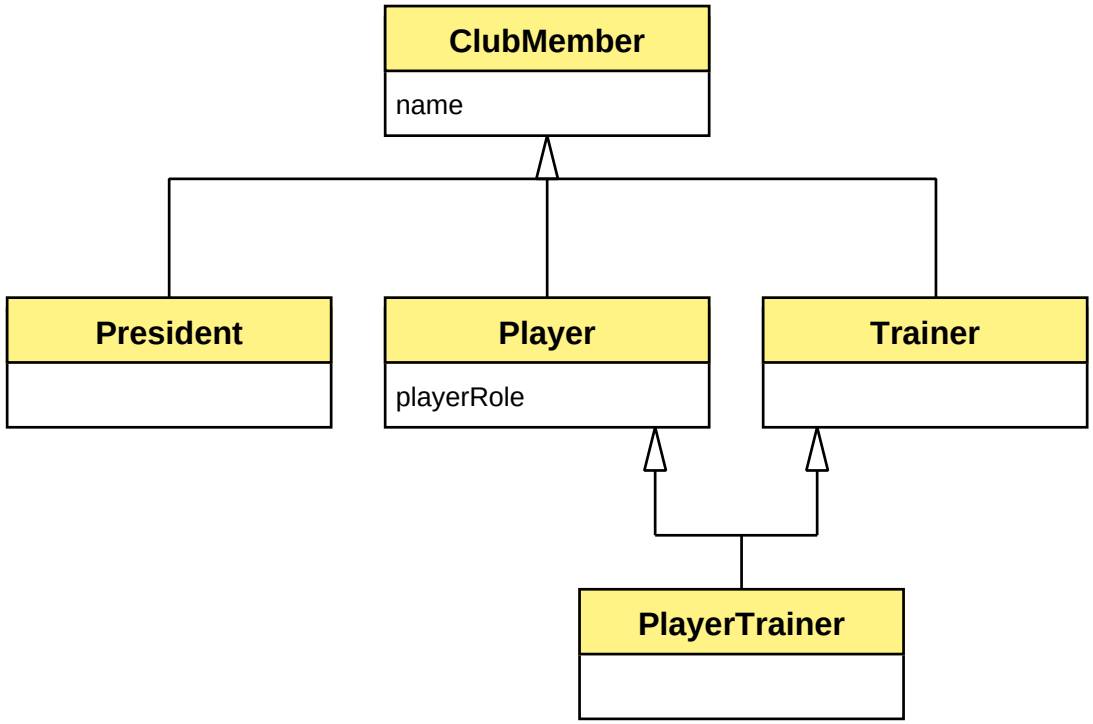



```
Class ClubMember -parameter {{name ""}}
Class Player -superclass ClubMember -parameter {{playerRole NONE}}
Class Trainer -superclass ClubMember
Class President -superclass ClubMember
```

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

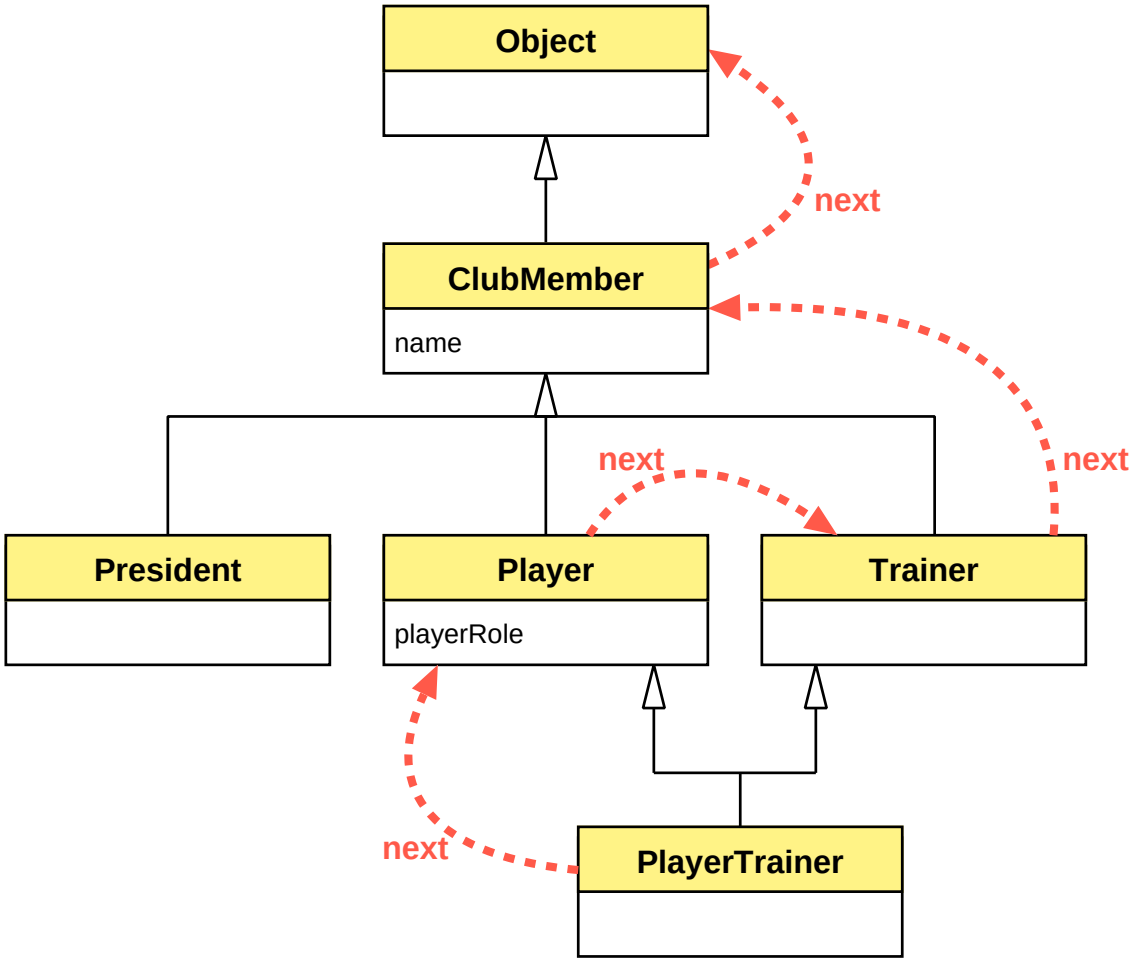
- **Each method call is performed on an object,**
- **If the method is not defined on the object, then the class and its superclasses are searched.**
- **If the method is found it may contain a `next` call.**
- **Then the "`next`" method on the class graph is searched and mixed into the current method.**
- **"`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"                      ;# Print ‘‘name’’
  next
}
Class Player -parameter {{playerRole NONE}}      ;# Subclass definition
Player instproc print {} {                       ;# Extended print operation
  [self] instvar playerRole
  puts "Player Role: $playerRole"                ;# Print player role
  next                                           ;# Call Superclass Implementation
}
```

Composability: `next` functions without naming the targeted superclass.

Method Chaining: Next Path for Player Trainer



Class-Path Linearization: Each class is visited once. Unambigous precedence order.

Dynamic Re-Classing

- **Dynamic classes and superclasses ⇒ Modeling life-cycle of objects.**
- **Example – Player becomes president:**

```
Player p -name "Franz Beckenbauer" \          ;# create player
  -playerRole PLAYER                          ;# life-cycle induces change
...                                           ;# reclassing to President
$fb class President
```

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

```
Player instproc class args {                  ;# Specializing class operation
  [self] unset playerRole                     ;# delete player role property
  next                                        ;# 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 for stadium
Class SoccerTeam                             ;# Soccer team class
SoccerTeam instproc init args {              ;# Constructor
  Stadium [self]::homeStadium                ;# Automatically aggregate stadium
  next                                       ;# New team instantiation
}                                            ;# Aggregate president
SoccerTeam bayern
President bayern::president \                ;# President leaves club -> disaggregate
  -name "Franz Beckenbauer"
bayern::president destroy
```

Object Aggregation

Aggregate with autoname:

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

Iterate over children:

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

Retrieving club name from parent:

```
ClubMember instproc getClubName {} {
  return [[[self] info parent] name]
}
```

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.**

```
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

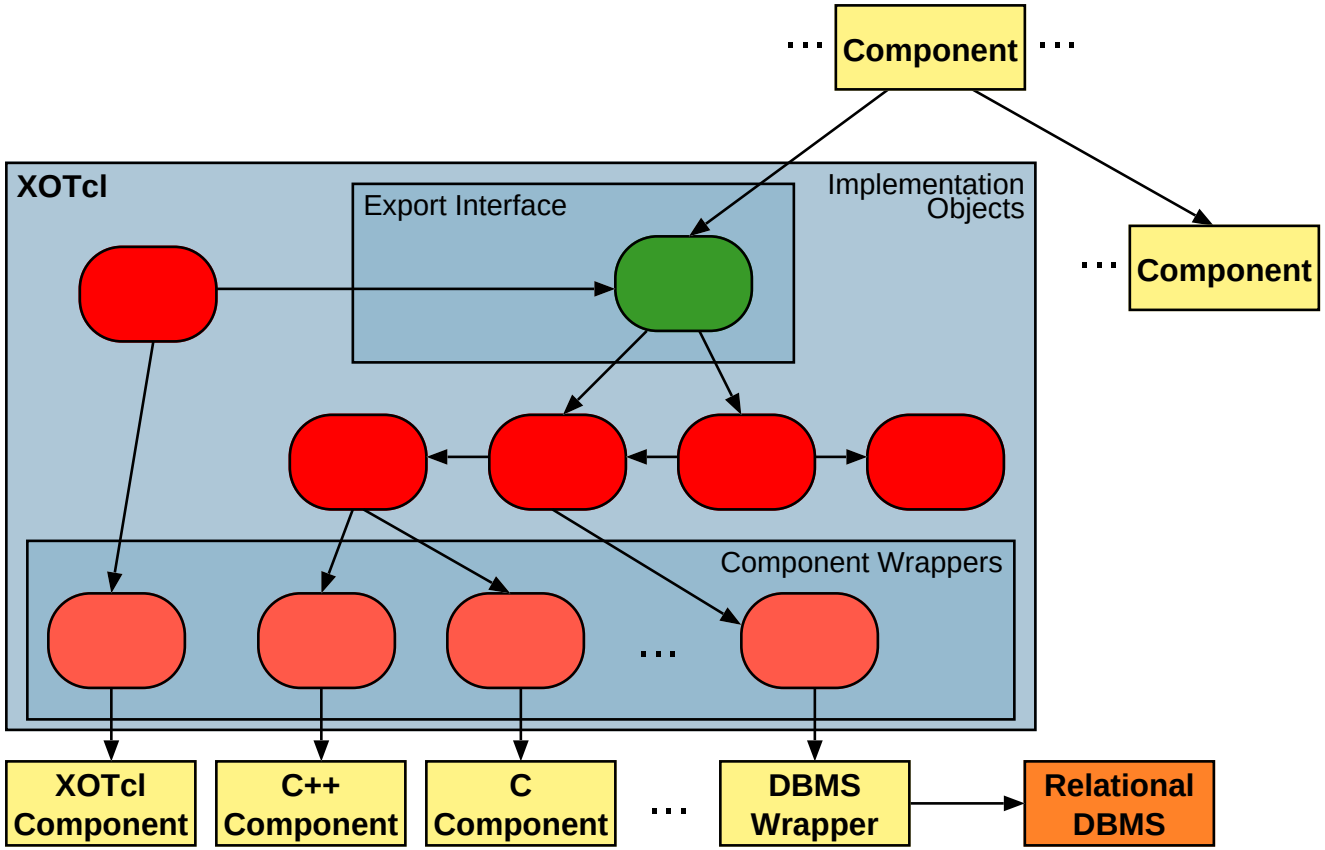
- **Component in XOTcl:**
  - Any assembly of several structures, like objects, classes, procedures, functions, etc.
  - Granularity: self-contained entity, i.e. subsystem or substantial part of a subsystem

- **Component has to declare its name and optional version information with:**
  `package provide` ***componentName ?version?***

- **Component can be loaded with:**
  `package require` ***componentName ?version?***

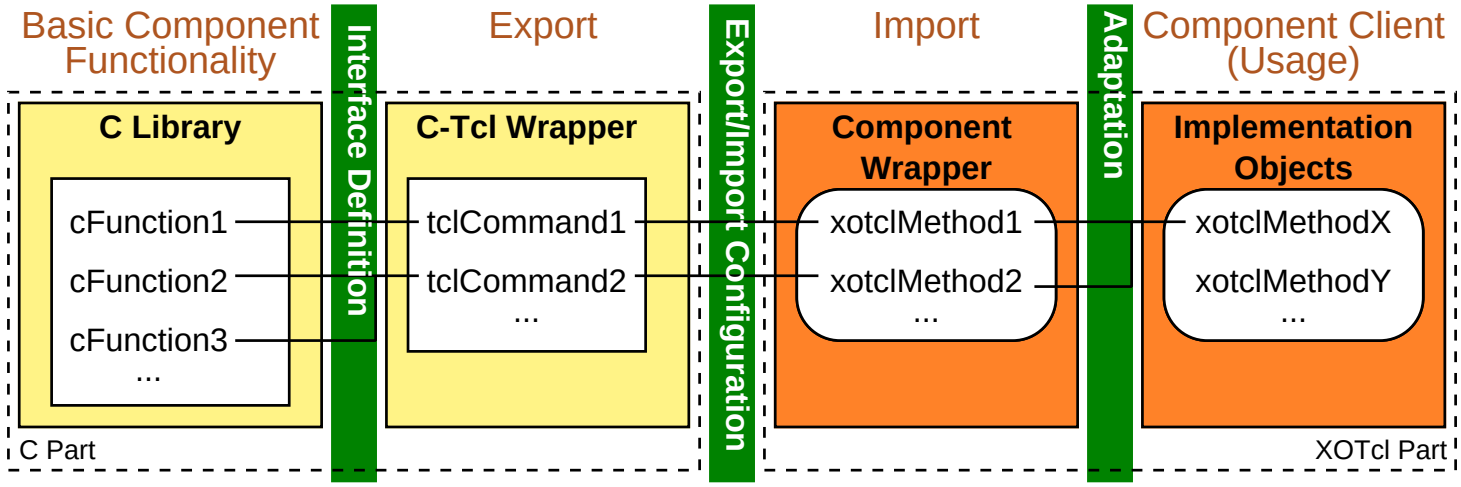
- **Automatic component indexing, tracking, and tracing.**

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



Three-Level Component Configuration: Make export and import explicit, first-class object → dynamic, runtime replaceability

Problems of a Pure Class-Based Implementation

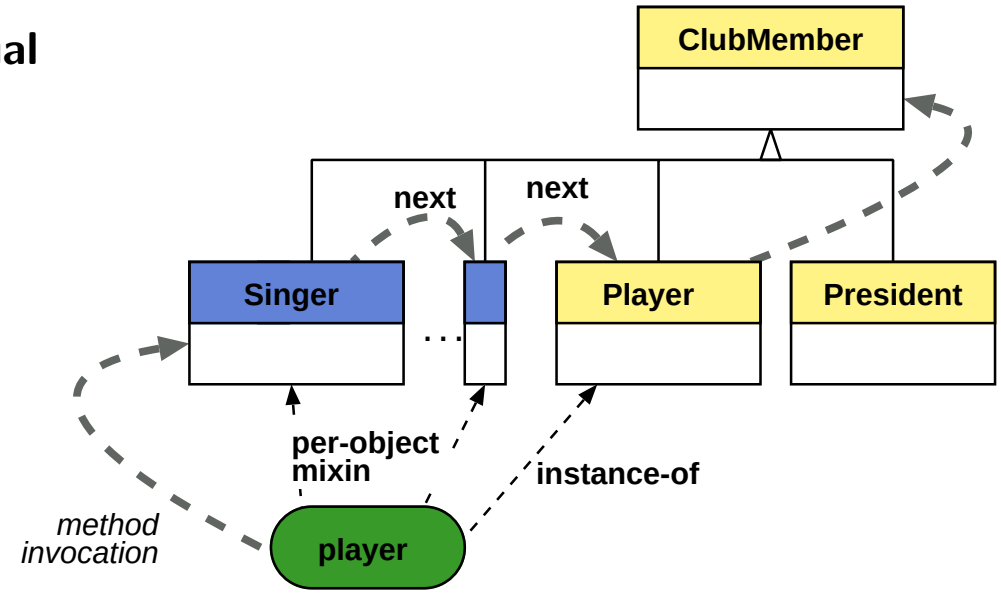
- **Transparency** – The client should not rely on concrete implementation details.
- **Decoration/Adaptation:**
  - Concerns that cross-cut the component wrapper hierarchy,
  - Object-specific component wrapper extensions or adaptations.
- **Coupling of Component and Wrapper**
  - Should appear as one runtime entity,
  - But: Should be decomposed in the implementation.
- **Component Loading** – Dynamical and Traceable

$\Rightarrow$ **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 \                          ;# Player object
  -name "Franz Beckenbauer"

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:

```
Class TransferObserver                          ;# Class definition
TransferObserver instproc transferPlayer \      ;# Transfer observer method
  {pname team} {
  puts "Player '$pname' is transfered."
  puts "Destination Team '[$team name]'"
  [self] set transfers($pname) $team
  next
}

SoccerTeam instmixin TransferObserver
                                                ;# Per-class mixin registration
bayernMunich transferPlayer \
  "Giovanne Elber" chelsea                      ;# Example transfer
```

Architectural Constraints

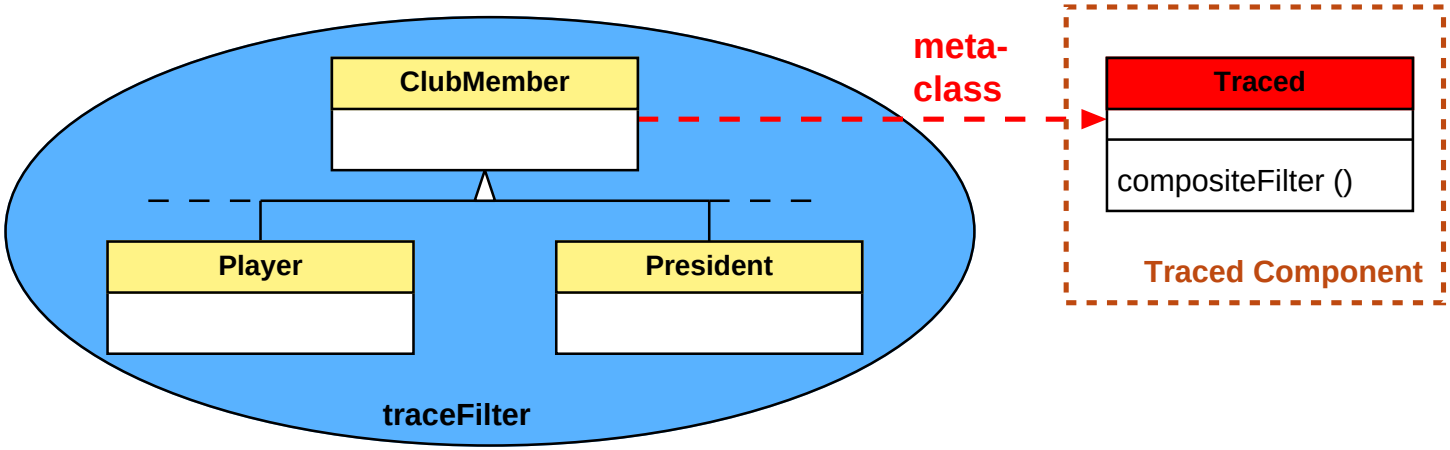
- **Restrict dynamic classes of sub-hierarchy to be static.**
- **Requests are split objects with C++ objects $\Rightarrow$ Dynamic classing is impossible.**

```
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                ;# Define component
...
Class Traced -superclass Class                   ;# Meta-class definition
Traced instproc traceFilter args {               ;# Trace filter method
  [[self] info regclass] instvar operations      ;# get traced operations
  set r [[self] info calledproc]                 ;# get callestack info

  if {[info exists operations($r)]} {            ;# check for registered operation
    puts stderr "CALL [self]->$r"                ;# print to stderr
  }
  return [next]                                  ;# perform target operation
}
Traced instproc init args {                      ;# Meta-class constructor
  [self] array set operations {}
  next                                           ;# Register filter
  [self] filterappend Traced::compositeFilter
}
```

Example: Composite Filter Usage

```
package require xotcl::Traced                    ;# Load component dynamically
...
Traced ClubMember \                              ;# Define traced class
  -addOperations {name ...}                      ;# Add traced operations

Class Player -superclass ClubMember              ;# Define different subclasses
Class President -superclass ClubMember           ;# => 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 $\Rightarrow$ 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

- 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

- . . .

Download ...