Lucent Technologies
Bell Labs Innovations

Using Java Reflection to Automate Extension Language Parsing

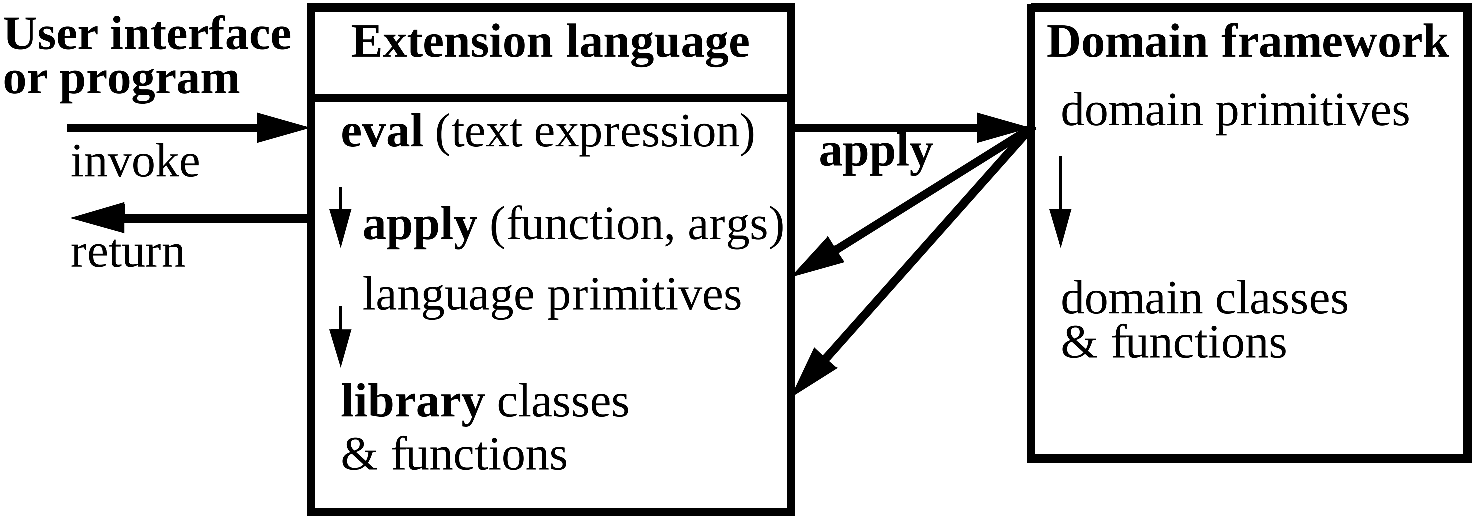
2nd Conference on Domain-Specific Languages

Dale Parson

Bell Laboratories / Lucent Technologies

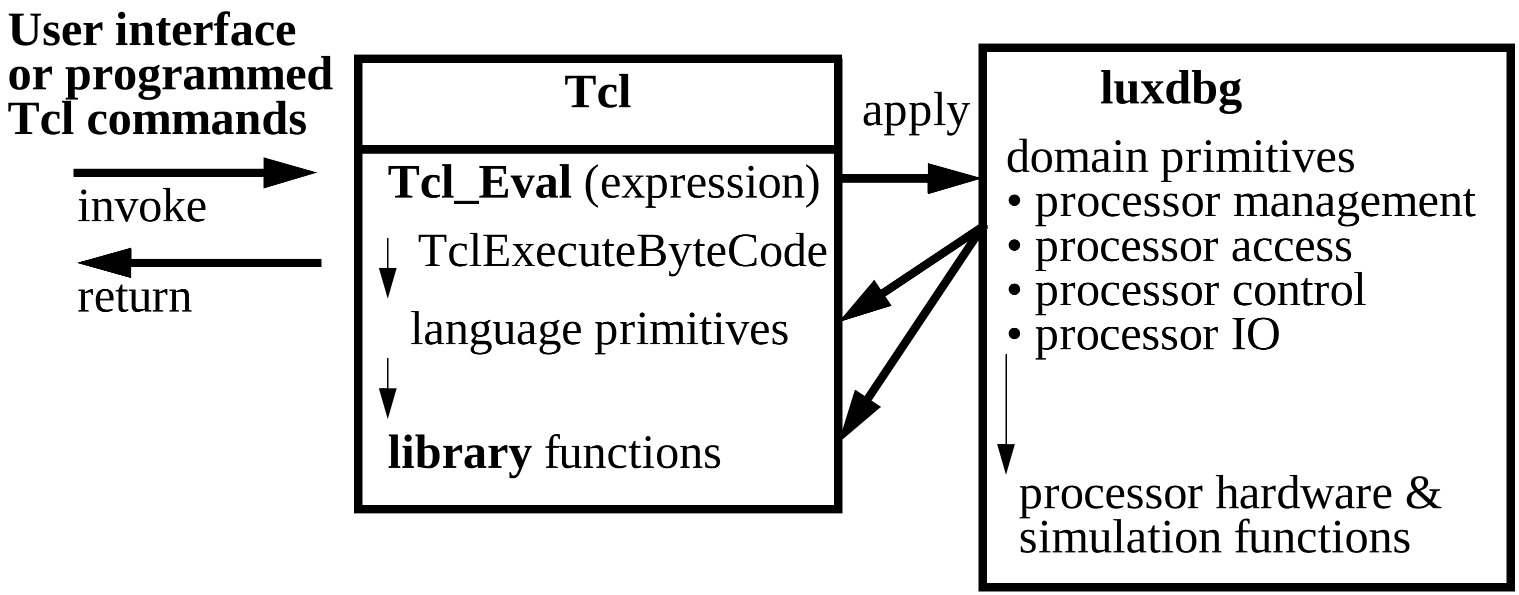
dparson@lucent.com

Domain framework - extension language system



- Framework adds domain-specific primitives to extension language instruction set
- Extension environment adds interpreted language capability to domain-specific framework
- Users extend the system by writing domain-oriented extension functions and auxiliary tools

Tcl - Luxdbg embedded system debugger



User interface or programmed Tcl commands

invoke

return

Tcl

Tcl_Eval (expression)

TclExecuteByteCode

language primitives

library functions

apply

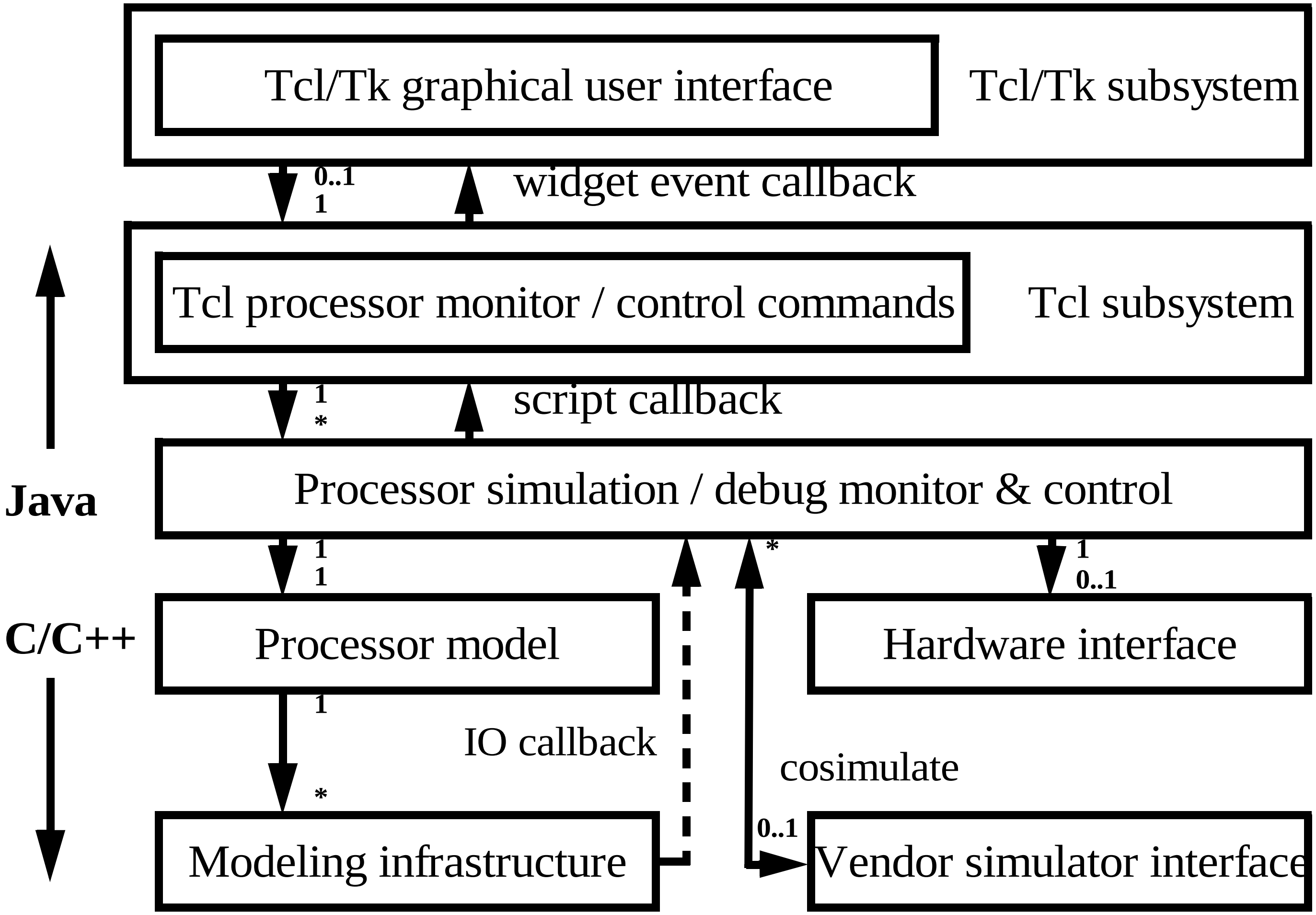
luxdbg

domain primitives

- processor management
- processor access
- processor control
- processor IO

processor hardware & simulation functions

- Luxdbg adds debugging and simulation primitives to Tcl instruction set
- Tcl adds interpreted extension language capability to debugger-simulator framework
- User scripts can manipulate and synchronize multiple processors, and create Tk GUI widgets


Tcl/Tk graphical user interface
Tcl/Tk subsystem
0..1
1
widget event callback
Tcl processor monitor / control commands
Tcl subsystem
1
*
script callback
Java
Processor simulation / debug monitor & control
1
1
*
1
0..1
C/C++
Processor model
Hardware interface
1
IO callback
cosimulate
*
0..1
Modeling infrastructure
Vendor simulator interface

Tcl-Luxdbg limitation 1:
Ad hoc primitive interface code

```
int awmpTclD::instStepi(ClientData clientData, Tcl_Interp *interp,
    int argc, char *argv[]) {
    awmpTclD::awmptclcInterp = interp ;
    int ret;
    if (argc != 1 && argc != 2) {
        Tcl_SetResult(interp, "usage: [ instanceName ] stepi [ count ]",
           TCL_STATIC);
        return(TCL_ERROR);
    }
    unsigned long stepcount = 1L ;
    if (argc == 2 && (!poorMansStrtoul(argv[1],&stepcount)
           || stepcount < 1L)) {
       Tcl_SetResult(interp, "bad stepi count: ",TCL_STATIC);
       Tcl_AppendResult(interp, argv[1], 0);
       return(TCL_ERROR);
    }
    . . .
}
```

- 1537 lines for 48 primitives (32 lines / primitive)

Tcl-Luxdbg limitation 2:
Hard-coded dependence on Tcl

- int Primitive(ClientData clientData, Tcl_Interp *interp, int argc, char *argv[])
- int TclPrim(ClientData clientData, Tcl_Interp *interp, int objc,Tcl_Obj *CONST objv[])
- Object ELK_Vararg_Primitive(int argc, Object *argv)
- int PyArg_ParseTuple(PyObject *argv, char *format,...)

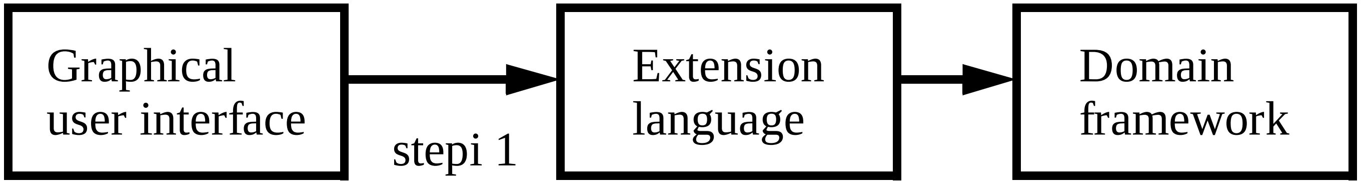
Each language passes an array of its **object type**.

We need a $type_x \times type_y \rightarrow type_{domain}$ **mapping**, where $type_x = \{\text{Tcl, ELK, Python, ...}\}$, $type_y = \{\text{integer, float, string, sequence}\}$, and $type_{domain}$ = set of Java types and classes.

Tcl-Luxdbg limitation 3:

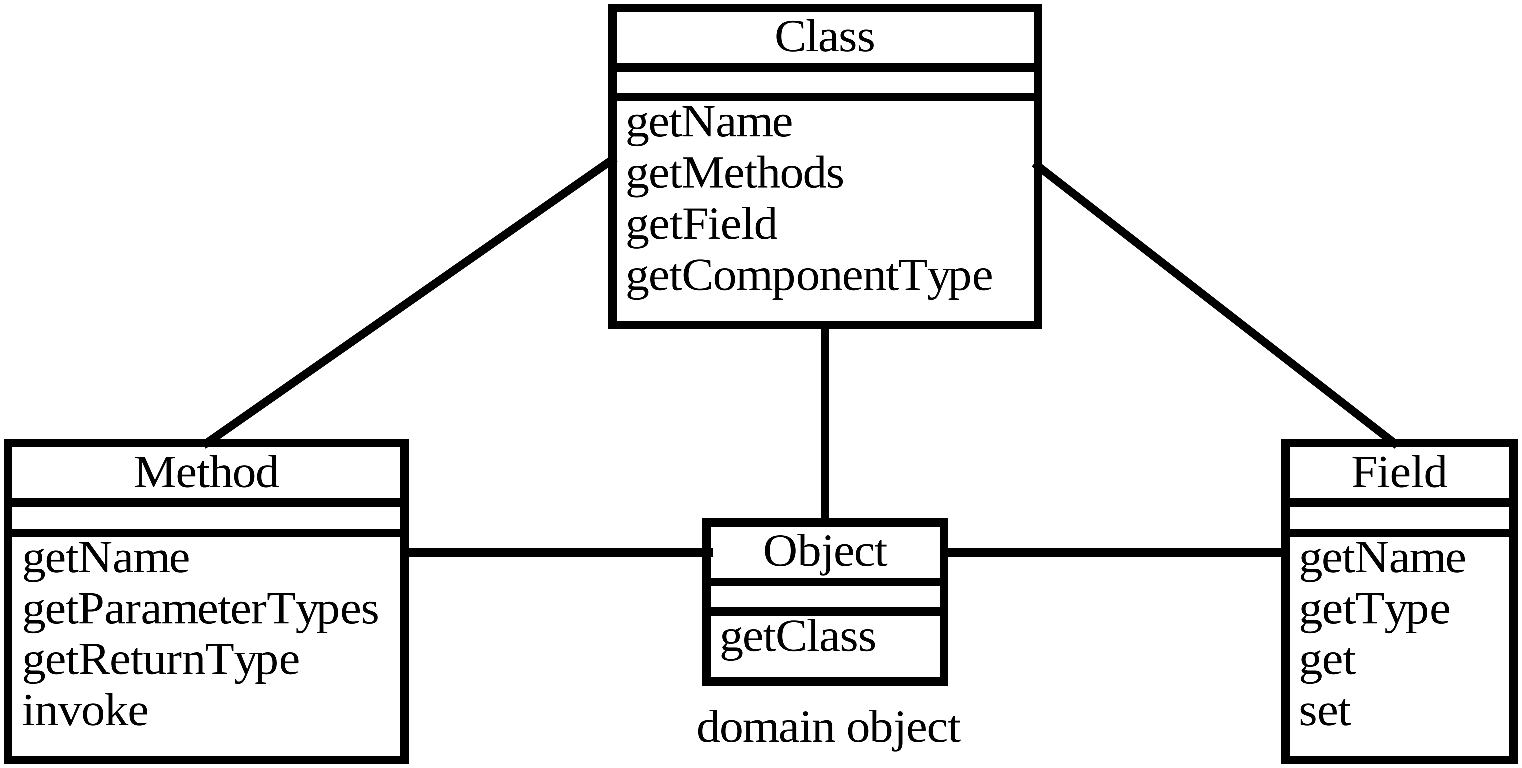
Unnecessary interpretation overhead

- Stereotyped commands invoke no extension language functions, control constructs or variables



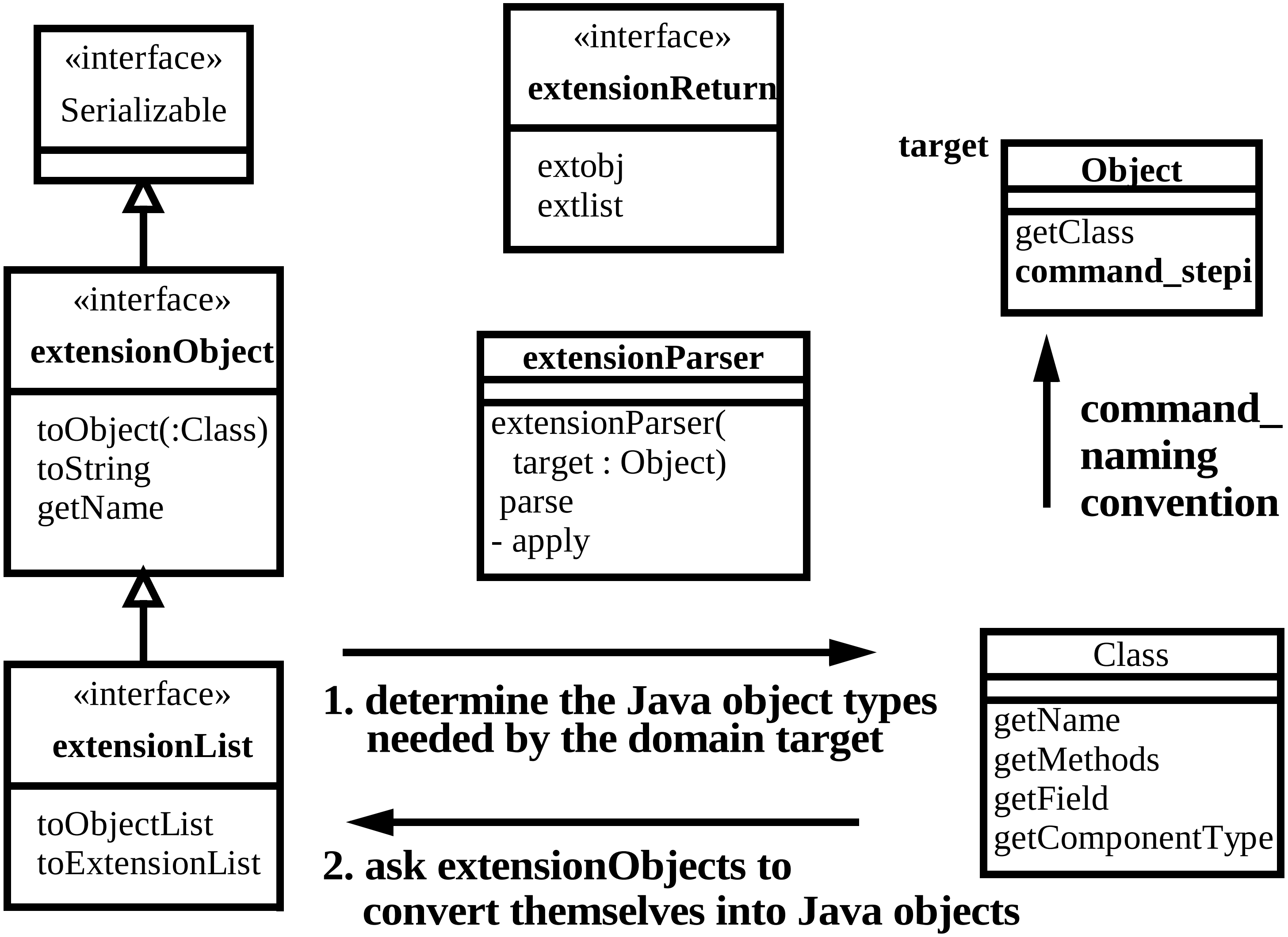
- Extension language provides uniform encoding
- Type signatures of domain primitives vary
- Avoid over-coupling GUI to command structure
- A “little interpreter” could apply GUI commands as function-argument invocations to a $\mathit{type}_{\mathit{domain}}$ command API

Java reflection provides the basis for a self-configuring “little interpreter”



- getClass for a domain object retrieves Java Class
- Class gives access to types & method invocation

Tcl-Luxdbg solution 1: reflection and a naming convention replace ad hoc interface code



«interface»
Serializable

«interface»
extensionObject

toObject(:Class)
toString
getName

«interface»
extensionList

toObjectList
toExtensionList

«interface»
extensionReturn

extobj
extlist

extensionParser

extensionParser(
  target : Object)
 parse
- apply

target

Object

getClass
command_stepi

command_
naming
convention

1. determine the Java object types needed by the domain target

2. ask extensionObjects to convert themselves into Java objects

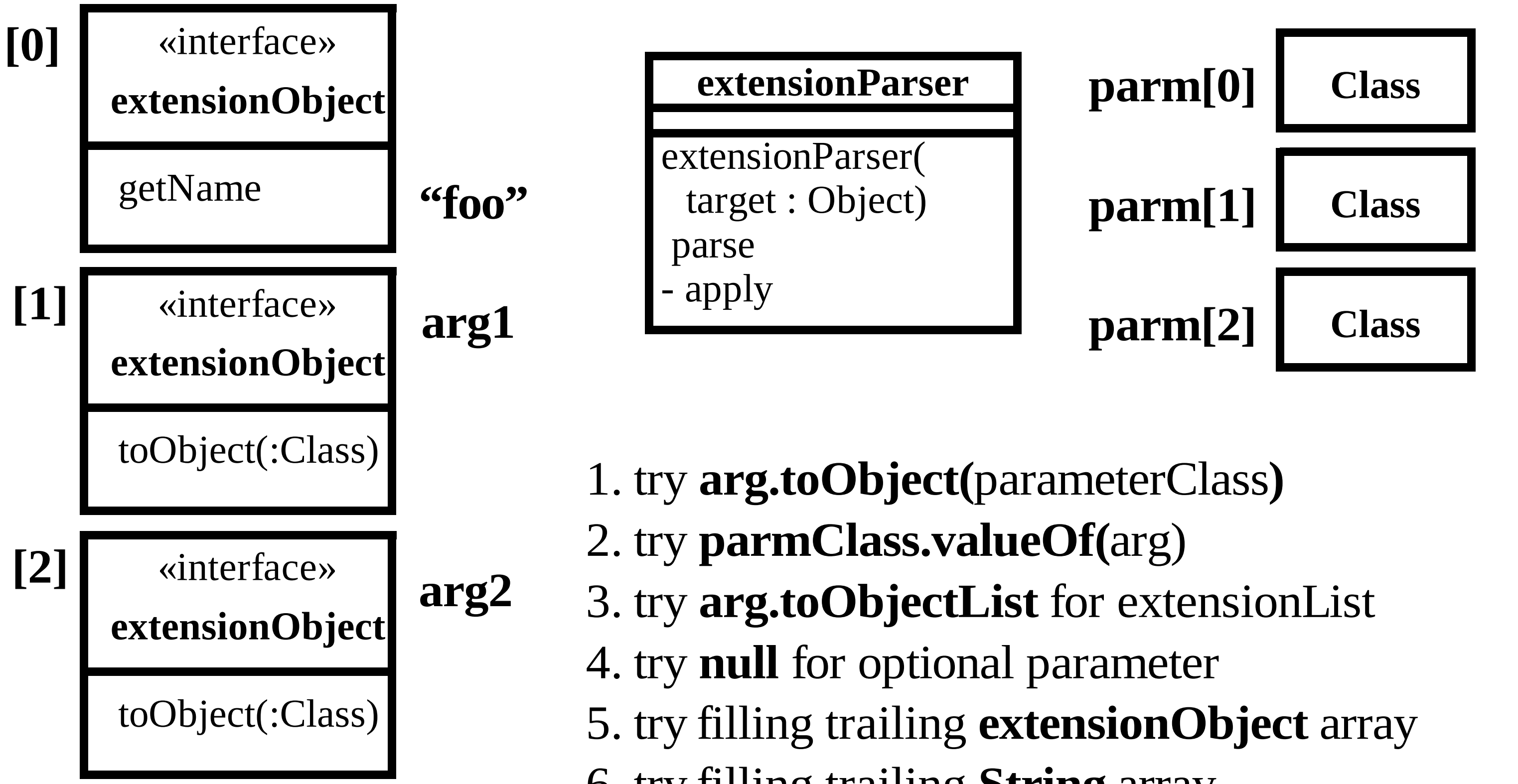
Class

getName
getMethods
getField
getComponentType

Tcl-Luxdbg solution 1: extensionParser

extension language apply ⟶

Class.getMethod(“command_foo”)
⟵ Method.getParameterTypes()

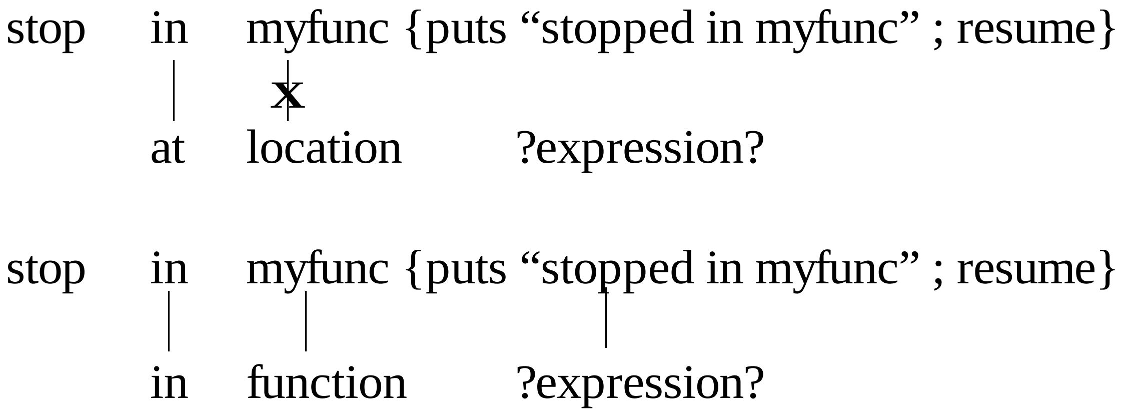


1. try **arg.toObject(**parameterClass**)**
2. try **parmClass.valueOf(**arg)
3. try **arg.toObjectList** for extensionList
4. try **null** for optional parameter
5. try filling trailing **extensionObject** array
6. try filling trailing **String** array

Tcl-Luxdbg solution 1: extensionParser

stop at location ?expression?
stop in function ?expression?

```
String command_stop(String keyword, int location,
     String expr)
String command_stop(String keyword,String function,
     String expr)
public static final int optional_stop_3[] = new int[1];
static { optional_stop_3[0] = 2 } ;
```



stop in myfunc {puts “stopped in myfunc” ; resume}

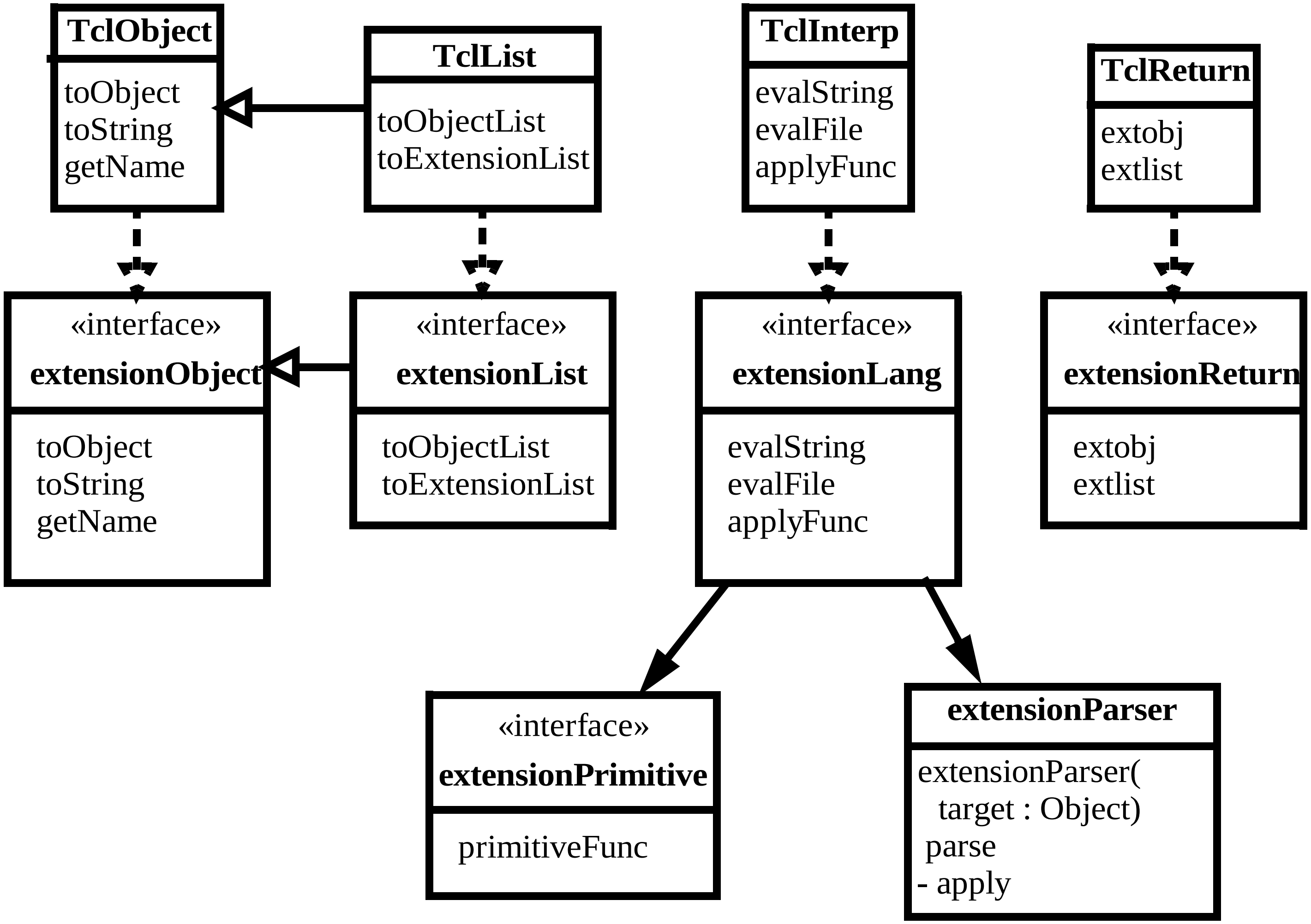
x

at location ?expression?

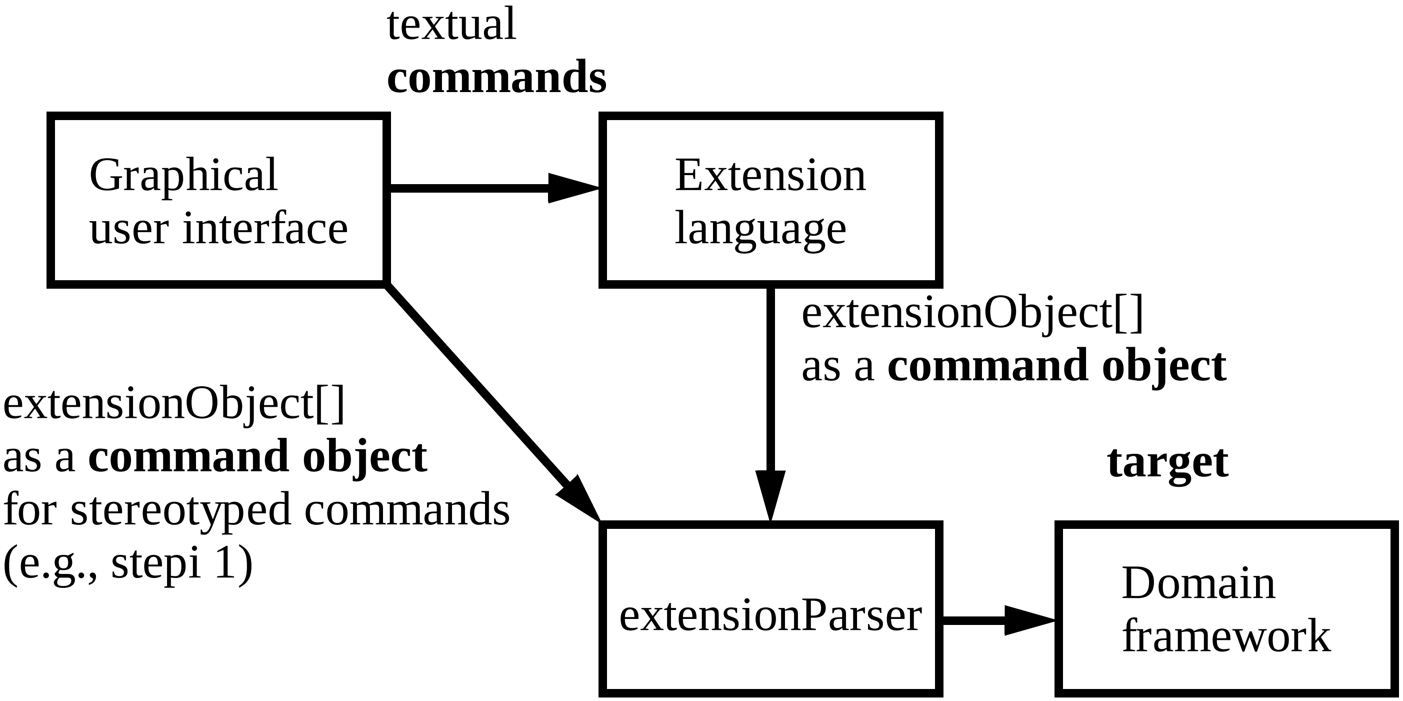
stop in myfunc {puts “stopped in myfunc” ; resume}

in function ?expression?

Tcl-Luxdbg solution 2: language as a parameter



Tcl-Luxdbg solution 3: command objects



- extensionParser is “little interpreter”
- extensionObject[] is command object that extensionParser applies

Solution 3: command object performance

Table 1: µSeconds-per-call for direct calls, command objects and interpreted expressions

test	direct	parsed command objects	Tcl 8.1.1 interpreter
argv, 1 method	0	42	399
argv, 50 methods	0	29	378
Tcl_Obj, 1 method	0	43	417
Tcl_Obj, 50 methods	0	36	391

- extensionParser.parse about 10% of the overhead of parsing stereotyped commands
- Tcl string interface marginally faster than Tcl object interface for loosely coupled Java system

Conclusions

- Extension language + domain framework = mutually extensible system
- Static language-framework linkage has limitations
- Java reflection + a method naming convention eliminate ad hoc code limitation
- Java reflection + interfaces + dynamic loading eliminate hard-coded language dependence
- Java reflection + command objects eliminate interpretation overhead for stereotyped commands
- Migration from C++ to Java has merit for interpreted command systems