



European Tcl/Tk User Meeting

Technische Universität Hamburg – Harburg
7. – 8. June 2001

Foreword

These are the proceedings of the Second European Tcl/Tk User Meeting, held the 7th. and 8th. of June in Hamburg.

For more information about this event or proceedings in PDF format see <http://www.tu-harburg.de/skf/tcltk>.

All copyrights *et cetera* remain at the original author, please contact them before using this material.

Carsten Zerbst (<mailto:Zerbst@Tu-Harburg.de>)

Contents

1	The (Active) State of Tcl	3
2	Why we use Tcl as strategic development platform.	13
3	LegacyTcl	33
4	XOTcl @ Work	39
5	Tcl for dynamic Web applications	61
6	tDOM	100
7	Game Scripting with Tcl	117
8	Generating test programs with TestMake	127
9	Creating generalised Tools for Database Access using Tcl/Tk	139
10	Using TCL as Middleware for Parallelizing Environment Development	144
11	Tcl on the iPaq	155
12	ENIÄK – High-level construction of user interfaces	161
13	mod dtcl web scripting with Tcl	169



The (Active) State of Tcl

Andreas Kupries

ActiveSTATE (<http://www.activestate.com>)

Programming for the People

Active STATE



The (Active) State of Tcl



Agenda



- **Introduction**
 - Andreas Kupries, ActiveState Corporation
- **What has happened since Tcl'Europe 2000**
- **Developments in the Tcl community**
- **Developments of the Tcl/Tk core**
- **Future directions**



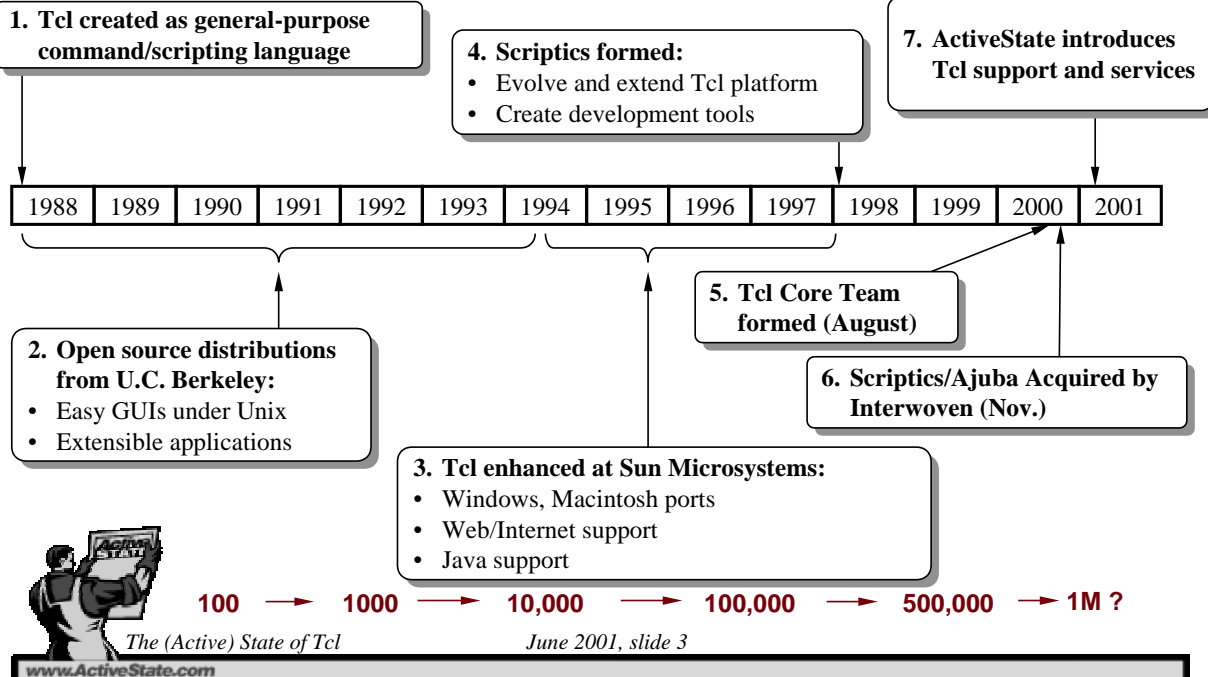
The (Active) State of Tcl

June 2001, slide 2

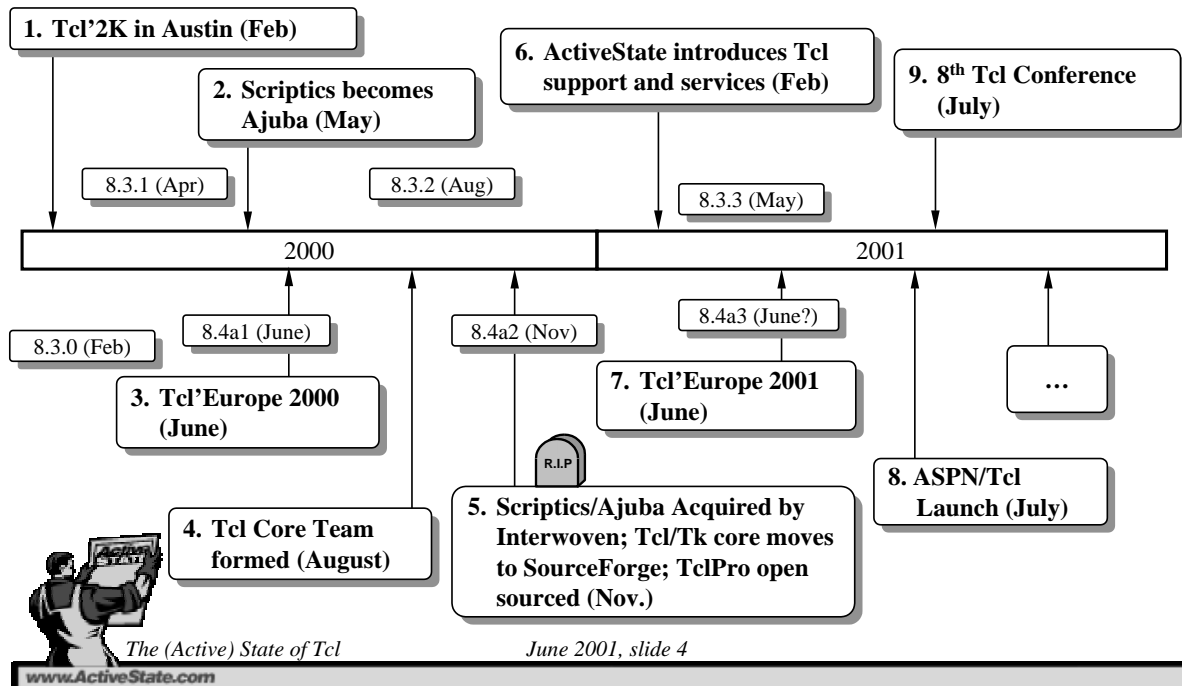
www.ActiveState.com



History of Tcl



Recent History of Tcl





Status as of Tcl'Europe 2000



- June 2000
- Tcl/Tk 8.4a1 recently released
- Tcl/Tk 8.3.1 was the stable version (now part of RedHat and SuSE standard distributions)
- Scriptics (now Ajuba) was focusing on B2B applications



The (Active) State of Tcl

June 2001, slide 5

www.ActiveState.com



Now... Tcl Core Team



- Formed in August 2000 with 14 charter members based on community voting:

Mo DeJong	Andreas Kupries
Donal Fellows	Karl Lehenbauer
Mark Harrison	Michael McLennan
D. Richard Hipp	Jan Nijtmans
Jeffrey Hobbs	John Ousterhout
George Howlett	Don Porter
Jim Ingham	Brent Welch

- Formed to collectively manage development of the core



The (Active) State of Tcl

June 2001, slide 6

www.ActiveState.com



TCT: TIP Initiatives



- Started TIP process for Tcl
 - <http://www.cs.man.ac.uk/fellowsd-bin/TIP/>
- **TIPs are intended to guide and document development on the core**
 - The focus is on new or changing features, not bugs
 - Voted on by the TCT following community discussion using the TYANNOTT process
- **Currently 34 TIPs (9 active project TIPs)**
- **Tcl/Tk maintainers are a separate group**
- **TCT discussion is open on the public mailing list:**
tcl-core@lists.sourceforge.net



The (Active) State of Tcl

June 2001, slide 7

www.ActiveState.com



Tcl/Tk Maintainers



- **Maintainers oversee a specific area of the core, as defined in TIP #16 for Tcl and TIP #23 for Tk**
- **They assist, but are not solely responsible for, fixing of bugs, adding documentation in their area**
- **They are responsible for reviewing code and approving code changes to their area**
- **Open to anyone willing to learn the core**
- **New volunteers always welcome**



The (Active) State of Tcl

June 2001, slide 8

www.ActiveState.com



The Maintainers...



➤ Tcl (TIP #24):

Allen Flick	Peter Spjuth	Todd Helfter	Jeff Hobbs
George Smith	Frédéric Bonnet	Kevin Griffin	Vince Darley
Chengye Mao	Jan Nijtmans	Donal Fellows	Mo DeJong

➤ Tk (TIP #30):

Daniel Steffen	Jim Ingham	Kevin Kenny	Jeff Hobbs
Miguel Sofer	Andreas Kupries	Rolf Schroedter	Vince Darley
Don Porter	Jan Nijtmans	Donal Fellows	Mo DeJong



The (Active) State of Tcl

June 2001, slide 9

www.ActiveState.com



Scriptics/Ajuba...



- **Scriptics became Ajuba Solutions in May 2000**
 - New focus as a B2B infrastructure company
- **Interwoven: content management company in need of B2B...**
 - Ajuba assimilated on Nov 1, 2000
 - Tcl/Tk moved to SourceForge:
 - <http://tcl.sf.net/>
 - TclPro open sourced:
 - <http://tclpro.sf.net/>
 - Further open source work not continued at Interwoven
 - Most other projects at Ajuba moved to SourceForge



The (Active) State of Tcl

June 2001, slide 10

www.ActiveState.com



Tcl/Tk at SourceForge



- **SourceForge provides a wealth of services for open source projects**
 - Bug and patch database
 - Mailing lists
 - CVS repositories
 - File server
 - Web pages
- **Managed by TCT and Tcl/Tk maintainers**



The (Active) State of Tcl

June 2001, slide 11

www.ActiveState.com



Tcl at ActiveState



- **Jeff Hobbs hired in Feb 2001**
- **Andreas Kupries follows soon after**
- **Wealth of scripting knowledge at ActiveState**
- **What ActiveState provides for Tcl:**
 - Improvements to open source Tcl core
 - High quality development tools
 - Komodo
 - ASPN/Tcl
 - Commercial support infrastructure
 - Professional services: training and consulting



The (Active) State of Tcl

June 2001, slide 12

www.ActiveState.com



In the Community...



- **The Tcl'ers Wiki has increased in activity:**
 - <http://www.purl.org/tcl/wiki>
 - Now with interactive chat
- **The Tcl Developer Xchange has moved:**
 - <http://www.purl.org/net/tclhome>
 - <http://tcl.ActiveState.com/>
- **Tcl-URL! continues to provide weekly news:**
 - <http://www.ddj.com/topics/tclurl/>
 - <http://tcl.ActiveState.com/tclurl/>
- **Lots of extension updates**



The (Active) State of Tcl

June 2001, slide 13

www.ActiveState.com



Tcl/Tk Today



- **Download rate steady (~30,000 / month)**
 - Windows: 60%
 - Unix: 45%
 - Mac: 5%
- **Only patch releases since last year**
- **Stable release now at 8.3.3**
 - Completely new I/O core (for 8.3.2)
 - High degree of stability
 - Improved locale support in Tk



The (Active) State of Tcl

June 2001, slide 14

www.ActiveState.com



Tcl/Tk 8.4



- **Experimental release, now at 8.4a2**
- **Still in feature-add mode**
- **New ‘spinbox’ widget**
- **Several minor core feature enhancements**
- **Significant work on performance**
 - Near or better than 8.0, with unicode and thread safety.
- **Several TIPs in the pipeline**
 - New virtual file system code
 - ‘lset’ command



TEA 2.0

The (Active) State of Tcl

June 2001, slide 15

www.ActiveState.com



Future Directions



- **The core is guided by community input**
 - Anyone can write a TIP
 - Anyone can be a core maintainer
- **What issues are most pressing?**
- **Open discussion**



The (Active) State of Tcl

June 2001, slide 16

www.ActiveState.com



Tcl Roadmap Poll



- Improve Tcl performance
- Archive file support (.jar/.zip)
- Standard libraries
- Unix binary distributions
- Tcl Installer
- Stand-alone executable support
- ...
- ...
- ...
- Smaller, more modular core
- Drag & Drop
- Windows Tk Performance
- Printing support
- Tk abstraction layer
- Megawidgets
(roll your own)
- New Widgets
- ...
- ...
- ...



The (Active) State of Tcl

June 2001, slide 17

www.ActiveState.com



Why we use Tcl as strategic development platform.

Andrej Vckovski (<mailto://andrej.vckovski@netcetera.ch>)
netcetera (<http://www.netcetara.ch>)

Tcl/Tk

Why we use Tcl as strategic development platform.

Andrej Vckovski
andrej.vckovski@netcetera.ch

Tcl User Meeting Hamburg, 2001

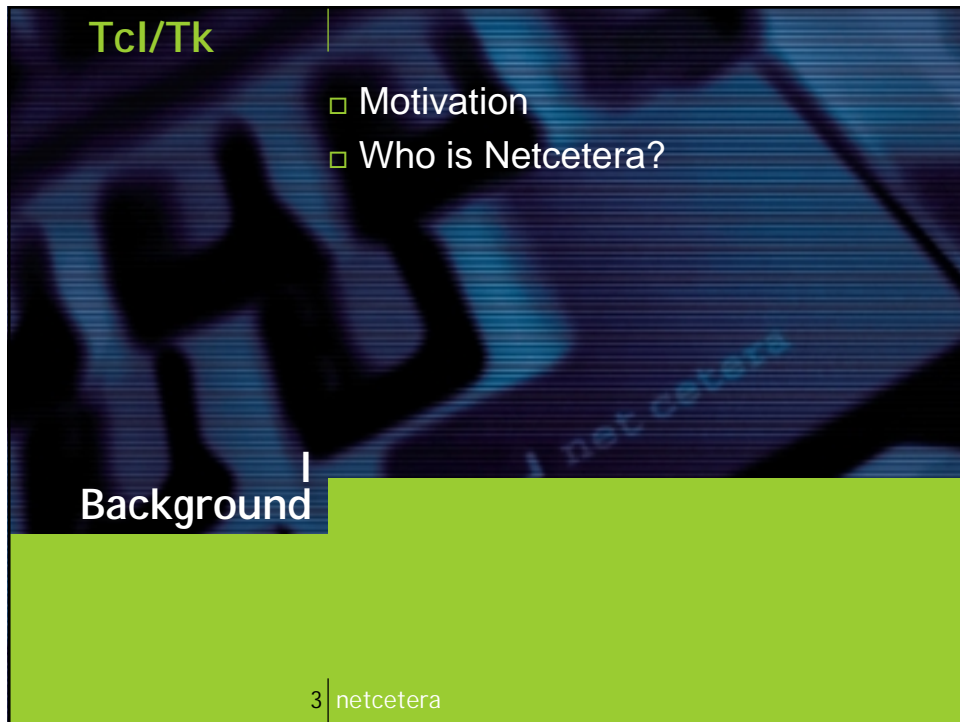
netcetera

Tcl/Tk

- I Background
- II What are we using Tcl for?
- III Selling Tcl and comparing it.
- IV Some Conclusions

Overview

2 netcetera

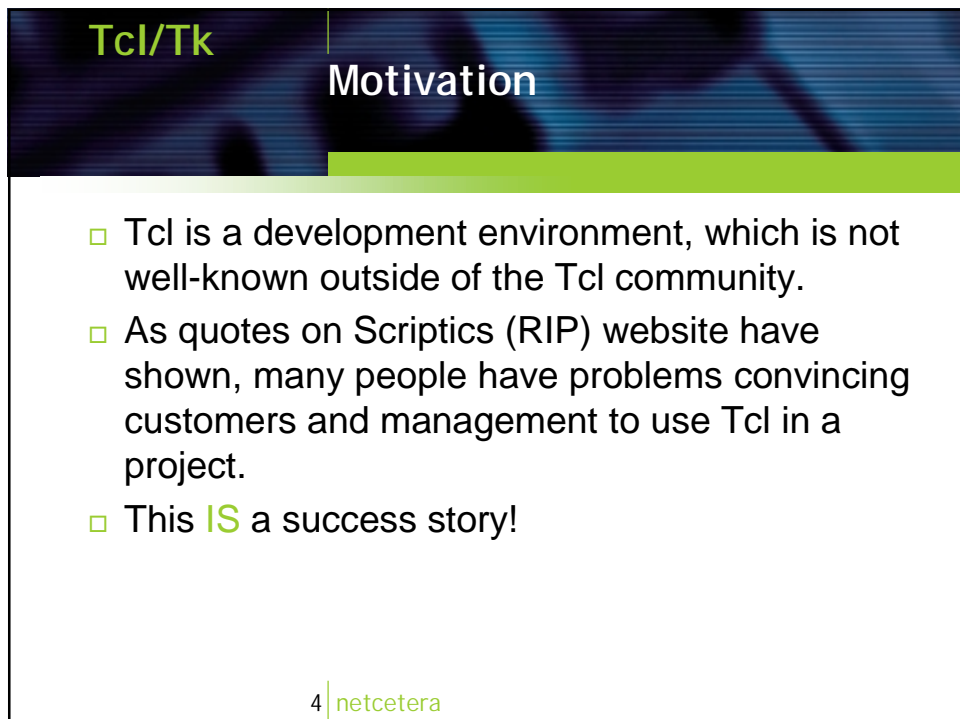


Tcl/Tk

- Motivation
- Who is Netcetera?

Background

3 | netcetera



Tcl/Tk

Motivation

- Tcl is a development environment, which is not well-known outside of the Tcl community.
- As quotes on Scriptics (RIP) website have shown, many people have problems convincing customers and management to use Tcl in a project.
- This **IS** a success story!

4 | netcetera

Tcl/Tk

Who is Netcetera?

- Founded 1995/1996 as spin-off of University of Zurich with a team of 5
- Initial business idea:
 - Do everything related to Internet, except Internet access (ISP) and graphical web site design.
 - Use our expertise in Unix and related technologies.

5 | netcetera

Tcl/Tk

Who is Netcetera?

- Netcetera 2001:
 - approx. 60 people (40 engineers, 8 operations and marketing/communication, 12 project management)
 - Mostly application development for customer projects
 - No strategic focus on financial industry, but that's what happens if you are located in Zurich, Switzerland.

6 | netcetera

Tcl/Tk

Who is Netcetera?

- How do we work?
 - Most development happens on Unix (Solaris, Linux and some AIX)
 - Most engineers have Windows NT boxes with an X-Server
 - Most development is in Tcl (variants ...) and Java
 - The IDE we are using is mostly Emacs with Makefiles

7 | [netcetera](#)

Tcl/Tk

A few figures

	lines of code	source files	loc/file
Tcl	794 529	3701	214
Java	337 228	2519	270
C/C++	153 365	568	134
Total	1 285 122	6788	

8 | [netcetera](#)

Tcl/Tk

- Our initial motivation for Tcl
- Various areas where we use Tcl
- A few applications explained

II What are we using Tcl for?

9 | netcetera

Tcl/Tk

Why did we start using Tcl?

- Our first application was an online order entry for a chain of pizza delivery services.
- There was experience with CGI programming using Tcl, but:
- We believed, we need to do it in C++ for „professionalism“ and performance.
- Still, we embedded a Tcl interpreter for the only reason to have a configuration file parser (configuration was a Tcl script that was evaluated and that did set some variables).

10 | netcetera

Tcl/Tk

Why did we start using Tcl?

- The C++-approach was extended to a C++-based framework for CGI applications (**web++**), now heavily using the very flexible configuration mechanism provided by the embedded Tcl interpreter.
- Developers soon started to place most business logic into the configuration files, because it was much easier.
- As a result, we designed **websh**, which was a Tcl-based scripting environment using the web++-framework.

11 | netcetera

Tcl/Tk

Why did we start using Tcl?

- To maintain a certain focus and critical mass, we decided to limit the number of platforms and languages we're using (mainly Java and Tcl).
- So we started doing also other stuff in Tcl (internal applications, ...)
- Today, we use Tcl for:

12 | netcetera

Tcl/Tk

... web applications on demand

- Dynamic Web sites
- E-Commerce Applications, Shops
- Banking / Stock Quotes
- Community Systems (e.g., researcher network for European Space Agency missions)
- Intranet Applications

13 | netcetera

Tcl/Tk

... CORBA Services

- Based on a framework (TACO), we can easily develop CORBA services where the business logic is implemented in Tcl.
- A few high-volume and mission-critical system in Switzerland largest bank use this approach.
- Framework also automatically generates test clients that allow scripted test suites.

14 | netcetera

Tcl/Tk

... internal tools

- Most internal tools for our IT operations, e.g.:
 - E-mail-archiving
 - Backup control
 - DNS administration
 - Network and system monitoring
 - Intranet and documentation tools
 - CRM, data warehouse, and problem reporting applications
 - ...

15 | netcetera

Tcl/Tk

... process support

- Tools that automate/control aspects of our software development process, such as:
 - Tools for version and configuration management (on top of CVS)
 - Tools for controlled builds (also for Java and C/C++ projects)
 - Documentation support („tclldoc“, similar to javadoc)

16 | netcetera

Tcl/Tk

... test automation

- Tcl is very useful for automated testing, regression testing:
 - for Tcl applications themselves
 - for Java using tclblend

17 | netcetera

Tcl/Tk

... gluing legacy applications

- The extensibility of Tcl makes it an ideal candidate to interface
 - legacy systems
 - strange hardware
 - anything that you know you need to interface but not yet what to do with it (i.e, make an extension that gives you access to the subsystem)
 - such extensions also provide a nice way to provide emulators (e.g., Tcl only implementation of the interface for development purposes)

18 | netcetera

Tcl/Tk

... products

- A few packaged products are built with Tcl:
 - **Netcetera SiteControl**, a site (and content) management system
 - **Netcetera PayControl**, a multi-channel payment integration platform
 - **wemlin**, a test automation product for web applications

19 | netcetera

Tcl/Tk

... embedding Tcl

- Websh3
 - Standalone interpreter, loadable extension and Apache module that allows easy development of web application. Open Source, written in C.
- Netcetera Cache Manager
 - A high-performance, multi-threaded cache manager that is built around Tcl's hash tables and works as a „main-memory“ database with many thousands of operations per second, written in C and C++.

20 | netcetera

Tcl/Tk

- Some Tcl „USP“ for
- Problems when „selling“ Tcl
- Comparing with Perl and Python

III Selling Tcl and comparing

21 | netcetera

Tcl/Tk

Some Tcl „USP“ for us

- Embeddable
Tcl interpreter can be added into existing applications, providing a great way for high level of configuration
- Thread safety
Interpreters can be used in threaded applications (e.g., web servers, CORBA services)
- Unicode
Is getting mandatory (XML, ...)

22 | netcetera

Tcl/Tk

Some Tcl „USP“ for us

- Event model
Very easy model for asynchronous applications (file events, timer events, ...).
- Platform independency
Tcl code runs on all of our supported platforms.
- Packaging
Tcl can be easily „packaged“ into a few, compiled files for easy and controlled delivery (but: it is getting harder).

23 | netcetera

Tcl/Tk

Intermezzo: How we package

- All Tcl source modules are concatenated into a simple file (we do not use Tcl's package mechanism).
- A small utility generates a shared library out of the file (the init code of the shared library optionally decrypts and then evaluates a large string).
- Standalone applications are started from a bourne-shell wrapper that exec's the interpreter and then loads the shared library.
- Motivation:
 - Have a minimal number of files to package and distribute.
 - Allow various different applications and versions on a system.

24 | netcetera

Tcl/Tk

Problems when „selling“ Tcl

- „Tcl is not object-oriented, modern SW development is object-oriented“
- My answer
 - There are object-systems for Tcl, but most importantly:
 - Object-orientation is not really a language feature but a design issue (even though, admittedly, „OO“ languages make an OO design more natural).
 - You can program object-oriented in Tcl as you can procedural in Java, but it needs more discipline.

25 | netcetera

Tcl/Tk

Problems when „selling“ Tcl

- „Tcl is slow“
- My answer
 - Most language comparisons show that Tcl is slower than other VHLL (very-high-level languages), but attention: Tcl 8.x has a few unique features that do have performance impacts (e.g., Unicode, thread safety, ...).
 - Performance is seldom really an issue (you always have the C-escape).

26 | netcetera

Tcl/Tk

Problems when „selling“ Tcl

- „Nobody uses it“
- My answer
 - Yes, usage numbers of, e.g., Perl or PHP are much higher.
 - But don't forget that Tcl exists within many commercial packages without telling you really.
 - Tcl is easy to learn.

27 | netcetera

Tcl/Tk

Problems when „selling“ Tcl

- „You cannot make high-quality application using a scripting (toy) language“
- My answer
 - Yes, type safety and many data types/structures help avoid some errors, but:
 - No 180°-turns when taking a prototype to a final product.
 - Usually faster implementation.
 - Changes of business logic and new features can be deployed at runtime.

28 | netcetera

Tcl/Tk

Problems when „selling“ Tcl

- „You must use Java. Everyone is using it and it can't be wrong“
- My answer
 - Ok!
 - Java is a great language, but not for everything.
 - I haven't yet seen a useful Web application environment for Java (Servlets and JSP suck).

29 | netcetera

Tcl/Tk

My comparison to Perl

- I am not a big Perl user, so my assessments might be wrong, it is an outside view:
 - (-) Perl has more ‚implicit‘ things and special variables
 - (+) Perl has better data structure support
 - (+) Perl has a better contributed sources archive
 - (-) Perl has more ways to do the same thing
 - (-) Perl is harder to package
 - (+/-) There is more ‚in the box‘ (e.g., system calls), but less portability

30 | netcetera

Tcl/Tk

My comparison to python

- I am not a big **Python** user, so my assessments might be wrong, it is an outside view:
 - (+) Python has a cool object model, and exception model
 - (-) Python has uncool intrinsic things (e.g., the underscores, meaningful white space)
 - (+) Python has many pre-packaged useful things
 - (-) Will be hard to package your application
 - (+/-) Seems to be very similar with regard to its power to Tcl
 - (-) Tcl has a long history, Python is still young

31 | netcetera

Tcl/Tk

- Would be switch?
- Our top wishes
- Summary

IV
Conclusions

32 | netcetera

Tcl/Tk

Would we switch?

- Currently, we are happy with Tcl, but are we **innovative**?
- The **best** language is the one that you know best. Is it?
- **How many languages** can a software engineering company bear?
- If we would choose now, we might also use **Python**, but not necessarily.

33 | netcetera

Tcl/Tk

Our top wishes for Tcl's future (unsorted)

- OO and Tcl
 - Embed one of the object models
- Documentation
 - Have a standardized way to document Tcl code à la javadoc.
- Java and Tcl
 - Speed up Jacl. Scripting environments for Java are a hot issue.
- Tools
 - Supplement the TclPro-tools with coverage analysis. Or better, a generic instrumentation engine.

34 | netcetera

Tcl/Tk

Summary

- We are using Tcl in our daily business, in mission critical applications and to earn money.
- Tcl is mature, robust and useful, but it lacks the hipness of Python, Pike and Java as well as their advocacy.

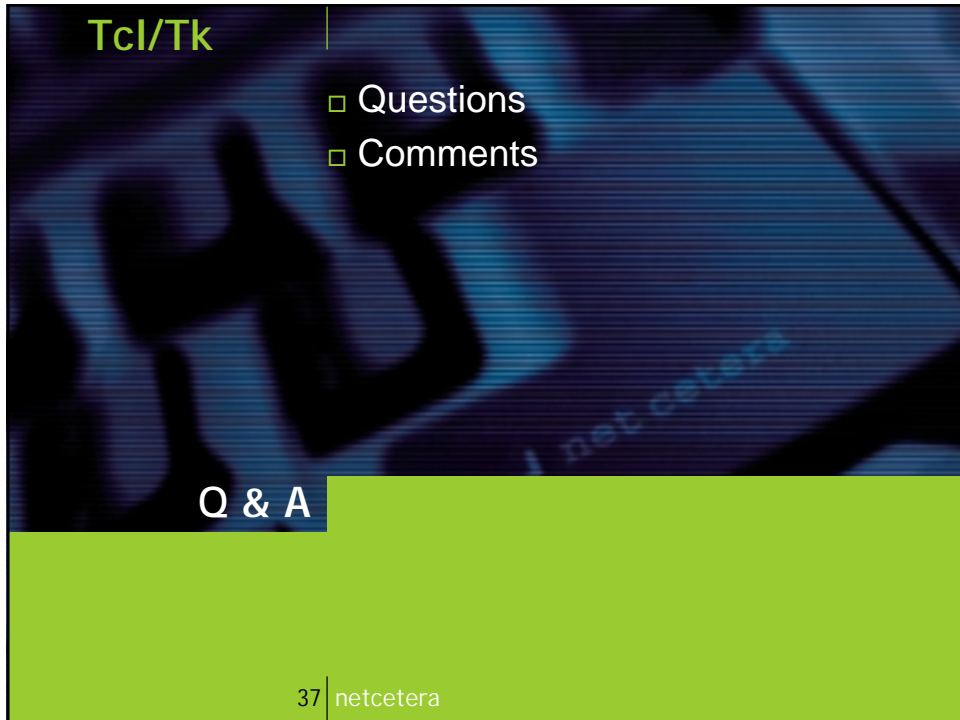
35 | netcetera

Tcl/Tk

Summary

- The context of application development (environments, tools, know-how sharing) is a very important thing. Sticking to a few such environments makes you much more productive, but there are high innovation risks.

36 | netcetera



Tcl/Tk

- Questions
- Comments

Q & A

37 | netcetera



Franco Violi (<mailto:fvioli@metodo.net>)

TclCobol**LegacyTcl**

TclCobol is an interface between the Cobol language and Tcl/Tk

- Tested with AcuCobol and MicroFocus Cobol
- Works in terminal mode (Tcl enabled)
- Works in graphical mode (Tk enabled)

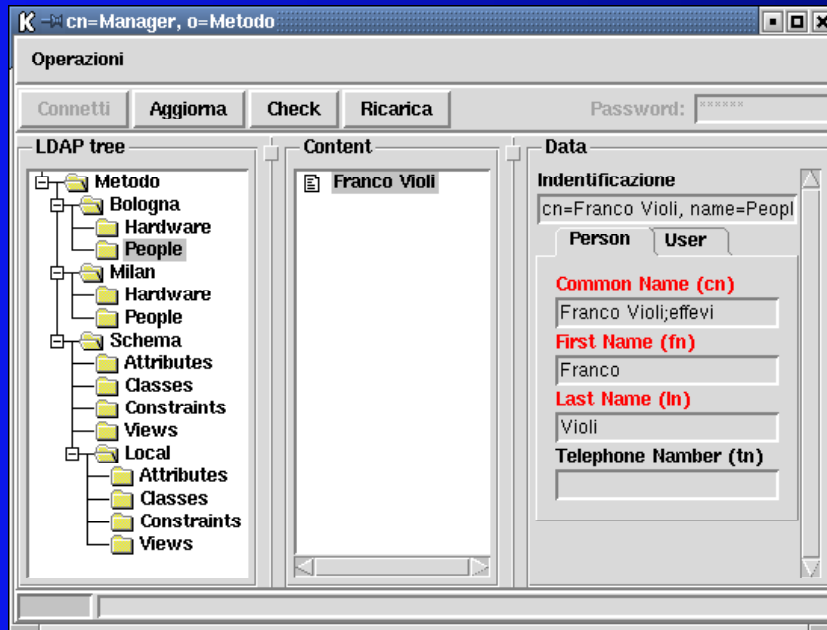
Franco Violi**fvioli@metodo.net****Abstract****LegacyTcl**

- TclCobol: an interface between Cobol and Tcl/Tk
- TclDirectory: a set of modules for administering a directory service
- TclCtree: an interface to the Faircom's DBMS engine
- TclDictionary: a set of tools for the data model definition
- TclDBMS: a set of tools for fast database application development
- LWidgets: a set of legacy widgets
- DBWidgets: a Tk toolbox for legacy application development

Franco Violi**fvioli@metodo.net**

A snapshot

LegacyTcl



Franco Violi

fvioli@metodo.net

TclDirectory

LegacyTcl

TclDirectory is a set of modules for administering a directory service

- A package to work on an LDAP server
- A package for running an object based dictionary schema
- An utility for checking the LDAP server consistency
- A Tk browser/mantainer for data maintenance

Requires

- Ldaptcl from Neosoft (included)
- openldap from openldap.org (1.2.x)
- BWidgets
- Itcl extension

Franco Violi

fvioli@metodo.net

Setting up a server**LegacyTcl**

Let's install TclDirectory

Check if you have openldap installed by typing 'ldapsearch':
if you have the command you are ready to go.

- Login as root and install the package wherever You want
- Go into the <install_directory>/tutor
- Modify the header of ldaptutor, as well of ../bin/ldaptree and put there your favorite wish program
 - i.e. #!/usr/bin/wish
- Type ./ldaptutor
- Create the configuration files and have a look to their contents

Franco Violi**fvioli@metodo.net****Some informations****LegacyTcl**

Some informations about the LDAP directory service

- It's something like a database, but
 - A column is an attribute
 - An attribute may have multiple values
 - A table is an objectclass
 - Columns can be shared between tables, that means objectclasses can share attributes
 - Rows are identified by their DN (distinguished name)
- DN is a comma separated list of ATTRIBUTE=VALUE
 - i.e. **cn=Franco Violi, ou=Developers, o=Metodo** is a valid DN on a server having a root directory named **o=Metodo**
- ... at the end, forget a RDMS, it is a directory !!!

Franco Violi**fvioli@metodo.net**

Run the browser**LegacyTcl**

Now choose Run TclDirectory

What TclDirectory does

- Check the binding DN
 - It uses -rootdn to get it, or
 - It tries to bind you using your POSIX attributes
- Check if the root node exist: if not, it creates the node
- Check if a valid schema definition exist: if not, it creates an empty schema
- It goes into browse/maintenance mode

Now, depending on your binding privileges, You are able to manage your own LDAP server.

Franco Violi**fvioli@metodo.net****Defining a root****LegacyTcl**

Now choose Reset the ldap database

- the root is o=Metodo
 - o= stay for organization, but you can use any other prefix
 - for example, ActiveDirectory uses dc=metodo.net, wich means a domain style convention
 - other common prefixes (or attributes) are
 - cn (Common Name)
 - ou (Organizational Unit)

Franco Violi**fvioli@metodo.net**

Understand the data model**LegacyTcl**

Choose Load the User Dictionary and have a look to the Local subtree

- Attributes
- Classes
- Constraints
- Views

Have also a look to the Schema subtree

YES, TclDirectory uses itself to work

Run the Do search button for the first contact with an LDAP database

Franco Violi**fvioli@metodo.net****Define the problem****LegacyTcl**

Let's think about our simple organization

- People (Person)
 - Using computers (User)
 - Not using computers
- Hardware
 - Computers (Computer)
 - Database servers (Database)
- People works in different locations
- Each location has a separate network

We say that the full organization is under the o=Metodo directory tree

Franco Violi**fvioli@metodo.net**



XOTcl @ Work

Gustaf Neumann
(mailto:Gustaf.Neumann@
wu-wien.ac.at)
Department of Information Systems
Vienna University of Economics

Uwe Zdun
(mailto:uwe.zdun@uni-essen.de)
Specification of Software Systems
University of Essen

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



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**



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 2

Motivation for XOTcl

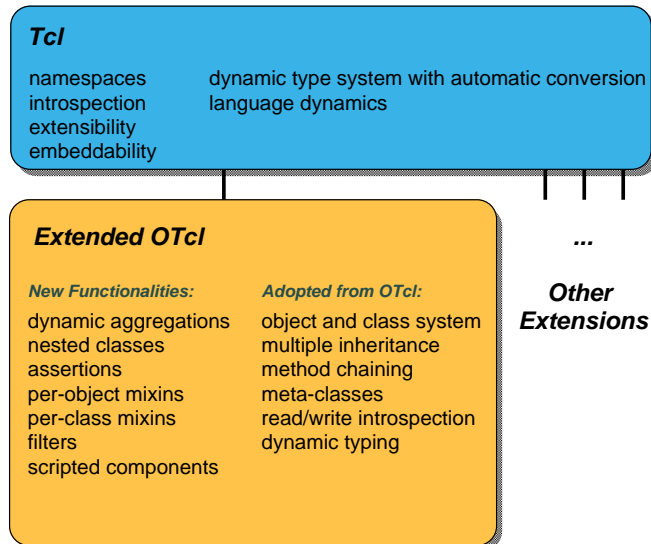
- ◆ **Extend the Tcl-Ideas to the OO-level.**
- ◆ **Just “glueing” is not enough! Goals are . . .**
 - Architectural support
 - Support for design patterns (e.g. adaptations, observers, facades, . . .)
 - Support for composition (and decomposition)
- ◆ **Provide flexibility rather than protection:**
 - 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*)



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 3

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" ;# Player 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,

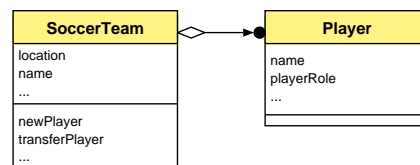
...

Object-Oriented Solution

◆ **Initial Design: Soccer team aggregates players.**

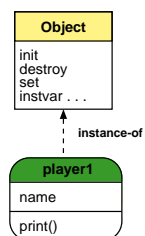
◆ **Used Concepts:**

- 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 var into proc scope
    puts "Name:      $name"                   ;# Print name to stdout
}

player1 print                                  ;# Call 'print'

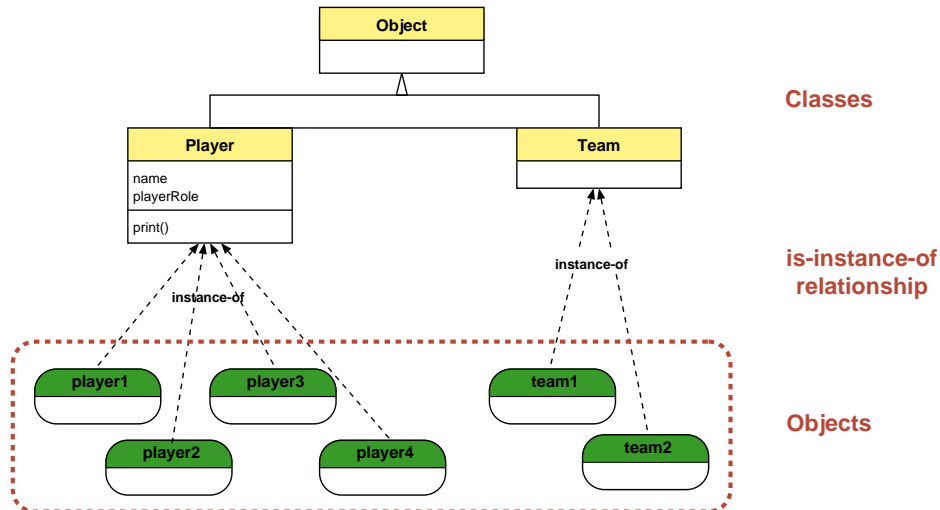
player1 destroy                                ;# And delete player object

```

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



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

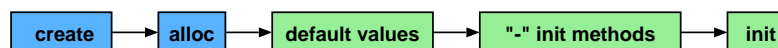
emile print                                              ;# Calling print operation
  
```

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
```

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:

<code>self</code>	current object name
<code>self class</code>	current class name
<code>self proc</code>	current proc/instproc name
<code>self callingobject</code>	calling class name
<code>self callingclass</code>	calling object name
<code>self callingproc</code>	calling proc/instproc name
...	...

- ◆ Example – Discriminating on calling object type:

```

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

```

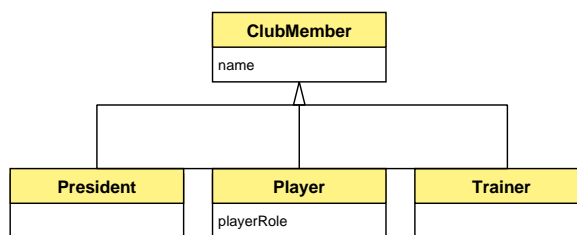


Gustaf Neumann, Uwe Zdun
University of Essen

Slide 16

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

```



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 17

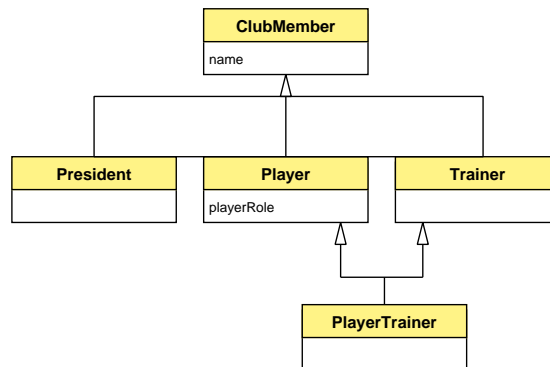
Multiple Inheritance

◆ **Multiple Inheritance** =
one class has more than
one superclass

◆ **Directed Acyclic Graph**

→ **Linearization**
Method Chaining

with



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

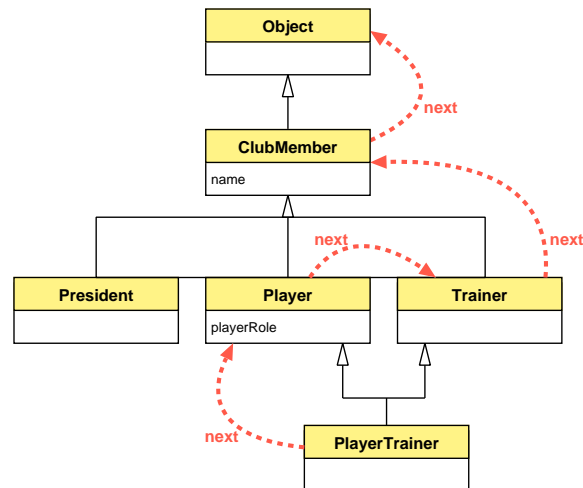
```

Class ClubMember -parameter {name}           ;# Class definition
ClubMember instproc print {} {                ;# Default print operation
  [self] instvar name                         ;# Print '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"             ;# Print player role
  next                                        ;# Call superclass implementation
}

```

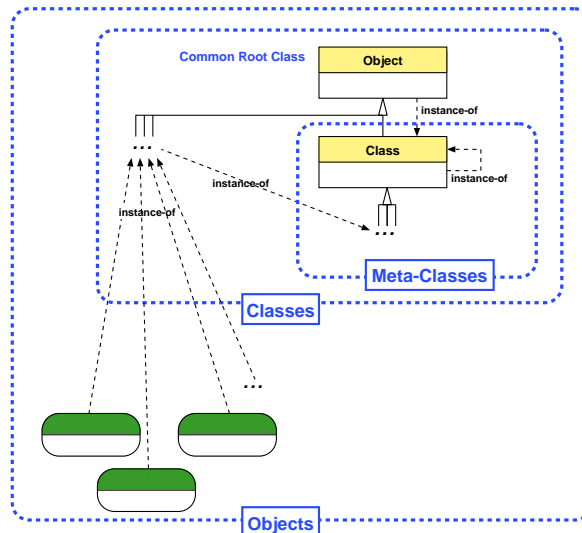
Composability: `next` functions without naming the targeted superclass.

Method Chaining: Next Path for Player Trainer



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

XOTcl Class and Object System



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
$fb class President                       ;# Reclassing to 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
}
SoccerTeam bayern                            ;# New team instantiation
President bayern::president \                ;# Aggregate president
    -name "Franz Beckenbauer"
bayern::president destroy                    ;# President leaves club -> disaggregate

```



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 24

Object Aggregation – Examples

Aggregate with autaname:

```

SoccerTeam instproc newPlayer args {
    eval Player [self]::[[self] autaname 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
}

```



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 25

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

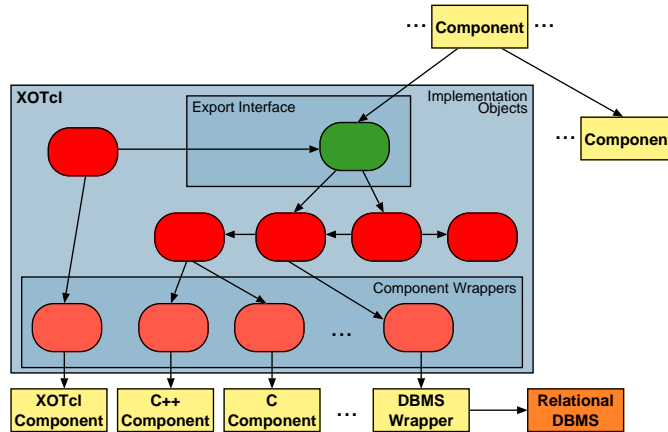
```
SoccerTeam instproc transferPlayer {playername destinationTeam} {
    foreach player [[self] info children] {
        if {[$player istype Player] && [$player name] == $playername} {
            $player move [set destinationTeam>::[$destinationTeam autname player%02d]
        }
    }
}
```

- ◆ **Object aggregation implies that the whole has responsibility of the life-time of the parts**

Dynamic Component Loading in XOTcl

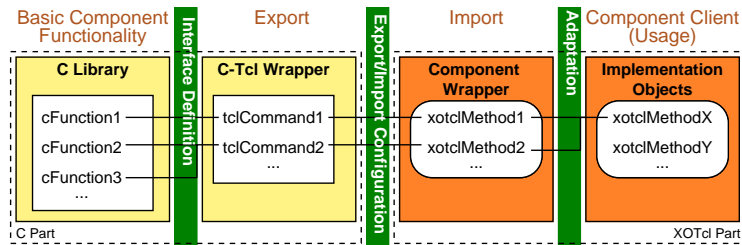
- ◆ **Component in XOTCL:**
 - Any assembly of several structures, like objects, classes, procedures, functions, etc.
 - Granularity: self-contained entity, i.e. subsystem or substantial part of a subsystem
- ◆ **Component has to declare its name and optional version information with:**
`package provide 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

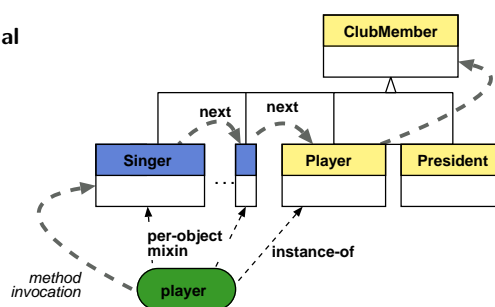
- ◆ **Transparency** – The client should not rely on concrete implementation details.
- ◆ **Decoration/Adaptation:**
 - Concerns that cross-cut the component wrapper hierarchy,
 - Object-specific component wrapper extensions or adaptations.
- ◆ **Coupling of Component and Wrapper**
 - Should appear as 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 \                                ;# Player object
  -name "Franz Beckenbauer"

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

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 \                         ;# Example transfer
  "Giovanne Elber" chelsea

```



Architectural Constraints

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

```

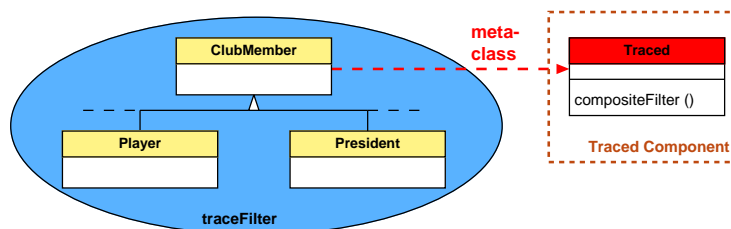
Class RestrictToSubClassOfRequest
RestrictToSubClassOfRequest instproc class args {
  set cl [[self] info class]
  next
  if {![self] istype Request}} {
    [self] class $cl
  }
}
Request instmixin RestrictToSubClassOfRequest

```

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
    set r [self calledproc]                     ;# Get callstack info
    if {[self regclass] 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
    [self] filterappend Traced::compositeFilter ;# Register filter
}

```

Example: Traced 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 & 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.)



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 40

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>



Gustaf Neumann, Uwe Zdun
University of Essen

Slide 41



Tcl for dynamic Web applications

Andrej Vckovski, (<mailto://andrej.vckovski@netcetera.ch>)
netcetera, (<http://www.netcetera.ch>)

Tcl/Web

Tcl for dynamic Web applications

Andrej Vckovski
andrej.vckovski@netcetera.ch

Tcl User Meeting Hamburg, 2001

netcetera

Tcl/Web

- I Introduction
- II General Features
- III Constructive Approaches
- IV Template-based Approaches
- V Tcl-based Environments
- VI A Short **websh3** Tutorial
- VII Q&A

Overview

2 | netcetera

Tcl/Web

- Definitions
- The foundation: HTTP
- Overview of different approaches

Introduction

3 | netcetera

Tcl/Web

What are interactive Web applications?

- Applications that generate HTML/WML/XML or other media types **on request**
- Content is not available as pre-built „files“
- Examples:
 - Internet banking
 - E-Commerce-applications
 - Web-Mail
 - you-name-it

4 | netcetera

Tcl/Web

What are software systems for interactive Web applications?

- Software that
 - supports development of such applications
 - provides a run-time environment
 - provides a tool-set to the developer

5 | netcetera

Tcl/Web

The basics: HTTP

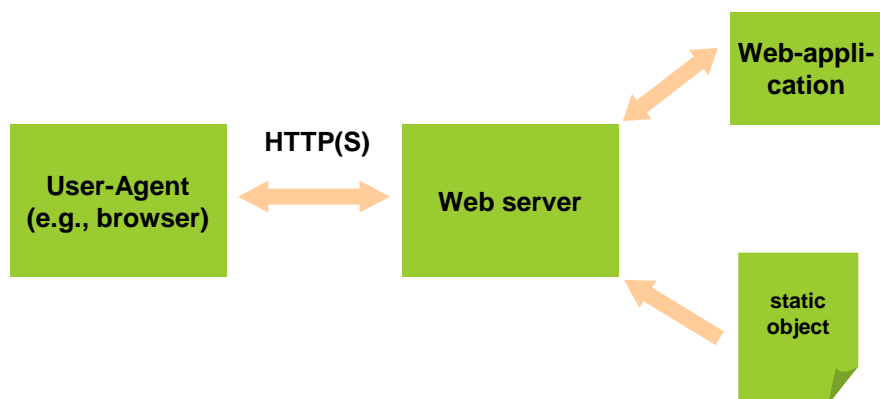
- Hyper-Text Transfer Protocol
- Simple, stateless **request/response** protocol on top of TCP
- **User-agent** (browser) sends a request
- **Web-server** sends a response:
 - status code
 - **MIME-object**:
 - HTML-text
 - image (GIF, JPEG, PNG)
 - applet
 - sound
 - Excel-file
 - ...

6 | netcetera

The basics: HTTP

- On the server, the MIME-object is either available as a static resource (a file, a database entry) or is dynamically created by a Web application.
- In most cases, the application generate HTML text.
- A request might contain additional data (e.g., data entered into a form). These data might be processed by the Web-server or Web application, respectively.
- The protocol is also available in a secure variant (HTTPS/SSL/TLS).

The basics: HTTP



Tcl/Web**Various approaches for Web applications**

- Template-based:
PHP, ASP, JSP, ColdFusion, ...
- „Constructive “:
Java Servlets, CGI, ...
- Web-server extensions
Apache Module, NSAPI, ISAPI, ...
- Custom HTTP servers

9 | netcetera

Tcl/Web

Protocol handling, session management, content creation, access to subsystems, authentication, logging, debug/test-support, URL manipulation, deployment-support, life-cycle management

**II
General
features**

10 | netcetera

Tcl/Web

HTTP protocol handling

- Decoding of form inputs, file-uploads
- Decoding of URLs
- Various HTTP-methods (GET, POST, PUT, ...)
- Control error codes/status
- Access to and generation of HTTP-headers
- Redirections

11 | netcetera

Tcl/Web

Session Management

- HTTP is state-less
- Applications nonetheless often need sessions
- Methods:
 - URL encoding (works always)
 - Cookies (works most (?) of the time)
 - hidden fields in forms
- Server-side persistence of **session-context**

12 | netcetera

Tcl/Web

Content creation

- Generate HTML/XML/WML-code
- Generate GIF/JPEG/PNG
- Generate PDF
- Separation from „design“-aspects if possible

13 | netcetera

Tcl/Web

Access to subsystems

- Database management systems
- CORBA-services
- Legacy-systems
- Enterprise Java Beans
- Hardware

14 | netcetera

Tcl/Web

Authentication/Security

- User authentication with username/password (HTTP Basic Authentication)
- Handling client-certificates when using SSL
- Defeat session-hijacking
- URL-encryption

15 | netcetera

Tcl/Web

Logging

- Logging of application activities:
 - Verbose in development and pilot phases
 - Terse in production
- Very important, because applications usually run 7x24h
- Control logging in running systems
- Consolidation/rotation when using multiple redundant systems

16 | netcetera

Tcl/Web

Debug/Test-support

- Support in development phases
- Load generators
- Interactive symbolic debuggers

17 | netcetera

Tcl/Web

URL manipulation

- Generation of URLs for self-referencing
- Supplement parameters for flow control
- Encryption

18 | netcetera

Tcl/Web

Deployment support

- Installation
- Activation
- Deactivation
- Configuration
- Different profiles for:
 - Development
 - Test
 - Production

19 | netcetera

Tcl/Web

Life-cycle management

- Versioning and configuration management
- Build-processes (controlled releases)
- Packaging

20 | netcetera

Tcl/Web

- Java servlets
- Web server extensions
- CGI
- FastCGI
- Custom Web Servers

III Constructive approaches

21 | netcetera

Tcl/Web

Java Servlets

- Java-class running in an application server
- Base for JSP
- Content needs to be generated „by hand“
- Advantage:
 - Performance
- Disadvantage:
 - „Low-level“

22 | netcetera

Tcl/Web

Web server extensions

- Most web servers offer APIs to extend their functionality with modules in C/C++/Java/...
- These modules often run in the server's address-space
- Examples:
 - Apache Module API
 - iPlanet/Netscape NSAPI
 - Microsoft Internet Information Server ISAPI
- Advantage:
 - Performance
- Disadvantage:
 - Robustness, complexity, „Low-level“

23 | netcetera

Tcl/Web

CGI

- Standardized interface that allows Web servers to call external applications for content creation.
- Advantage:
 - Robustness
 - Portability
- Disadvantage:
 - One process is spawned per request

24 | netcetera

Tcl/Web

FastCGI

- Similar to CGI, but a single process can handle multiple requests
- Advantage:
 - Performance
- Disadvantage:
 - Robustness, not „standardized“

25 | netcetera

Tcl/Web

Custom Web Servers

- Web servers that offer embedded functionalities, e.g.:
 - AOL Web Server
 - ...
- Existing applications that have an additional Web server component:
 - Printer software
 - Mail servers
 - ...

26 | netcetera

Tcl/Web

Scripting Languages (VHLL)

- For CGI, FastCGI and web server extensions, often scripting-languages (very-high-level languages) are used
- Definition of a **scripting language** is vague:
 - Often weak typing
 - Late compilation
 - Self-modifying code
- Examples:
 - Perl, Tcl/websh, Python, Pike, VisualBasic, PHP/ZEND

27 | netcetera

Tcl/Web

Scripting Languages (VHLL)

- Usually very expressive (few lines of code for lots of functionality)
- Robust
- Small memory-footprints (yes!)
- Usually slower than traditional, strong-typed and early-compiled languages such as C/C++ or even Java
- Frequently used in web application environments

28 | netcetera

Tcl/Web

- Basics
- Advantages and Disadvantages
- Tools

IV

Template-based approaches

29 | netcetera

Tcl/Web

Template basics

- Developer writes HTML-text with special markup which contains server-side executed code
- A template-processor parses these templates at runtime and substitutes application-generated HTML-text in these places
- Example:


```
.....
This is <strong>HTML</strong> with some
<%
set s "embedded"
puts $s %> code
...
```

30 | netcetera

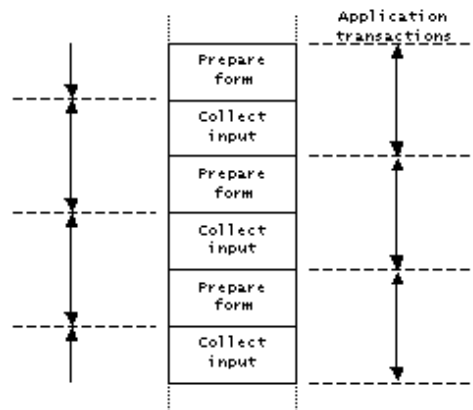
- Advantages
 - Looks very simple and straight-forward
 - Theory:
 - Someone who knows HTML can develop applications
 - Design work is separate from development work
 - Application programming is content management
 - But ...

- Disadvantages
 - Real applications have complex flows
 - High redundancy of repeatable design components (navigation, headers, ...)
 - No separation of graphical design and application development
 - One click (request) does not lead always to the same „page“

Tcl/Web

Advantages and Disadvantages

HTTP transactions vs. business-transactions:



33 netcetera

Tcl/Web

- Overview
- An incomplete list

V
Tcl-based
Enviornments



34 netcetera

Tcl/Web

Tcl-based Environments

- There are various commercial and non-commercial Tcl-based environments for web applications
- Some of them use Tcl but do not mention it (e.g., integration platforms that use Tcl as glue language)
- Steve Ball's **WebTcl complete** provides a good overview

35 | netcetera

Tcl/Web

An incomplete list

http://starbase.neosoft.com/~claird/comp.lang.tcl/server_side_tcl.html

- cgi.tcl (Don Libes)
- AOL-Webserver (AOL)
- NeoWebScript (NeoSoft)
- mod_dtcl (D. Welton)
- Kinetic Application Processor (M. Harrison)
- Storyserver (Vignette)
- VelociGen Application Server (Binary Evolution?)
- TclHTTPd (Brent Welch)
- websh3, mod_websh (Netcetera)
- ...

36 | netcetera

Tcl/Web

VI
A short
websh3
tutorial

37 | netcetera

Tcl/Web

38 | netcetera

a short websh tutorial

Simon Hefti

Andrej Vckovski

Ronnie Brunner

Netcetera AG, Zurich Switzerland

Tcl/Tk User Meeting Europe, June 7/8
2001, Hamburg

1 Agenda

- 1) what is webshell ?
- 2) web application development approaches
- 3) webshell tutorial
- 4) mod_websh
- 5) summary

2 what is webshell ?

web application development framework
reduced to the max

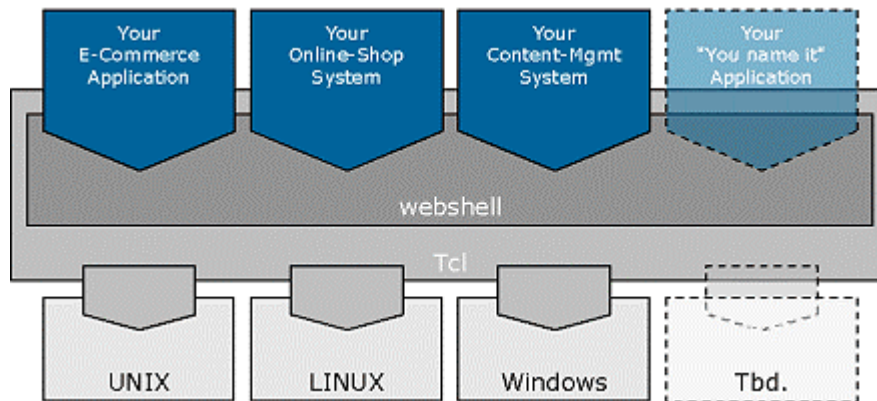


Figure 1 - System Design

webshell provides

a set of commands and data structures for quick and reliable web application development and deployment

2.1 webshell is

mod_websh

a dynamically loadable Apache module

websh3

a Tcl interpreter

libwebsh.so

a loadable Tcl extension

open source software

download from <http://websh.com>

quick reference

<http://websh.com/quickref.html>

2.2 key features

live demo

[shop application](#) | [code](#)

sessions

storage independent session handling

command dispatching

each page of your application is defined in the same script
command dispatching let's you jump from one page to the next

encryption

encrypt credit card numbers, URLs, sessions
webshell comes with an easily extensible interface for strong encryption

logging

your app is live 24 hr a day, year in, year out:
you *will* need logging

2.3 history

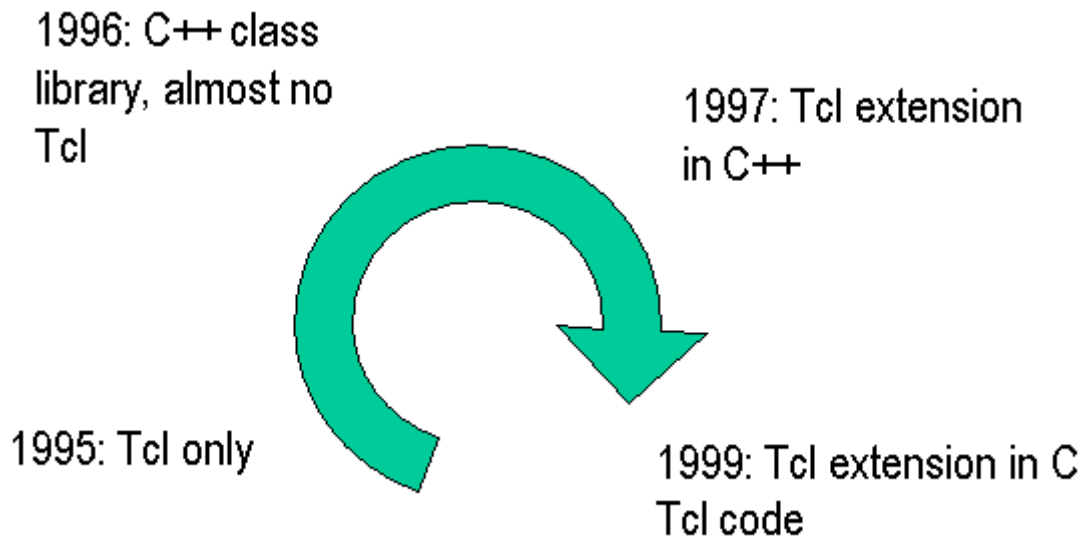


Figure 2 - History

2.4 various uses of webshell

webshell for everything

development in CGI

use the advantages of CGI (robust, portable, stateless) for web application development

deployment with mod_websh

use the advantages of embedded execution (performance) with mod_websh

operation/housekeeping with websh3

use your thorough know-how of the development language also to ensure operations and housekeeping

templates with mod_websh

complex applications: don't use ASP or JSP !
... but use mod_websh for dynamic HTML pages

3 selected features

- 1) session management
- 2) multi-state application
- 3) logging
- 4) access to form variables
- 5) other features

3.1 session management

session tracking

- group requests into a transaction
- session IDs are stored in URL or in cookies

generation of unique IDs

- built-in: file-based sequence number generator
- easily extensible

storage independent

- sessions also contain data
- but session-specific part of API is storage independent

cookies

- also supported - use if you must

3.2 multi-state application

URL generation

- including state, time stamps, and session tracking
- encrypted by default

dispatching

- automatic dispatching into various states within an application (single binary)

easy definition of control flow

- application in single binary - jump from one page to the next within your application

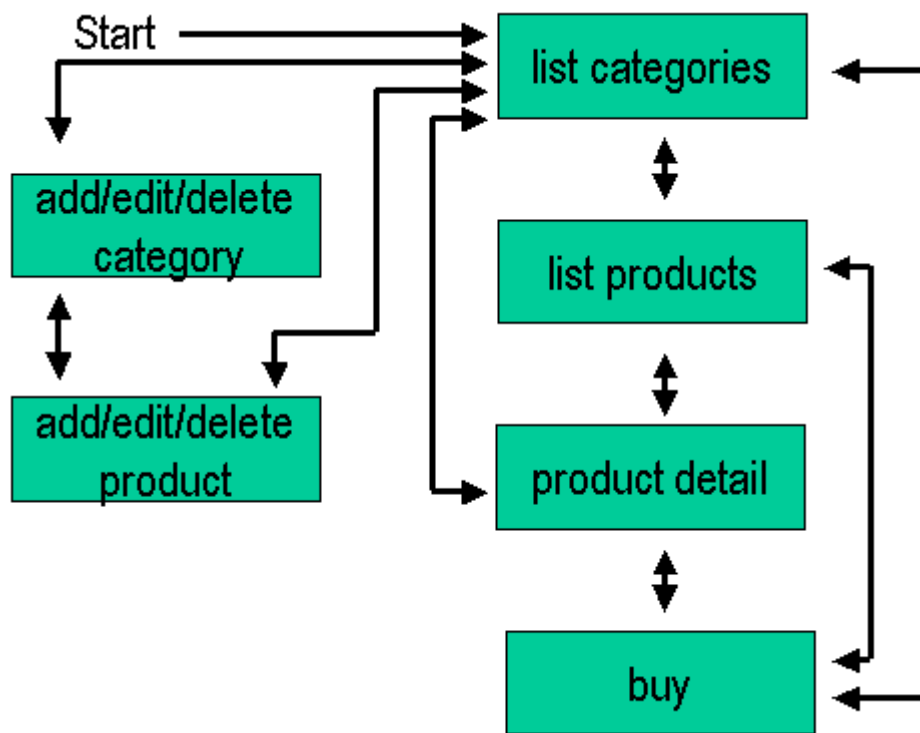


Figure 3 - states of the demo shop

3.3 logging

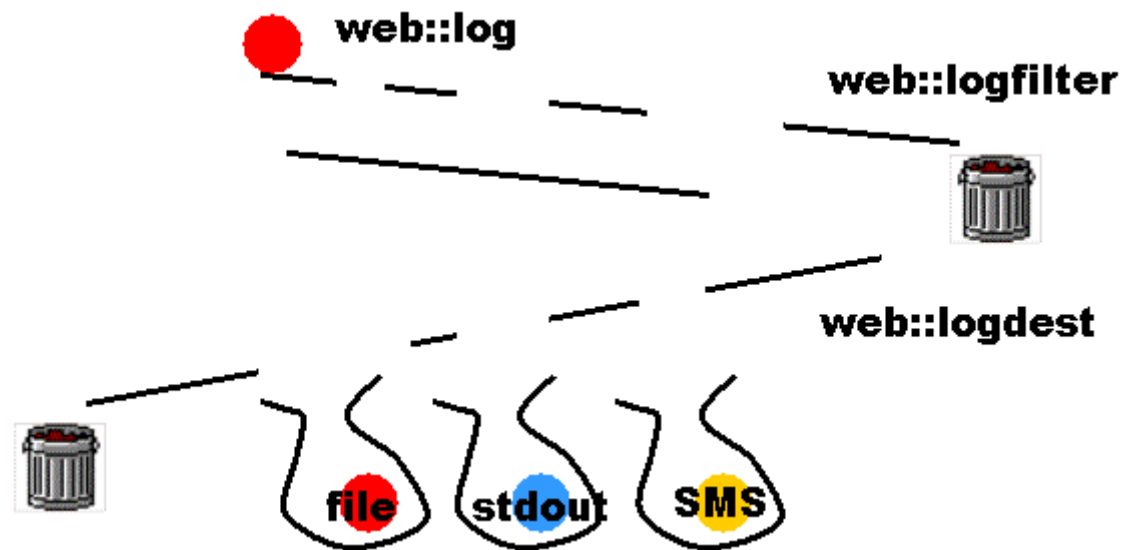


Figure 4 - logging

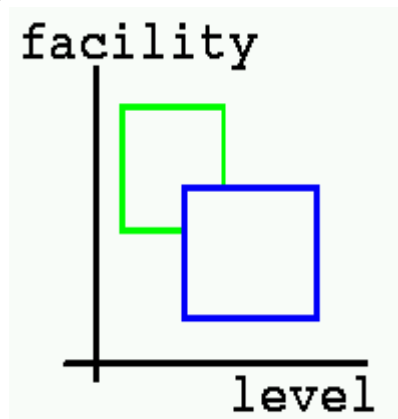


Figure 5 - log filters

3.4 other features

encryption

E-commerce *needs* encryption
designed for extensibility

buffered output

have as many output channels as you want

memory channels

read from and/or write to variables

control over output

set HTTP headers and error code, use encoding

versatile, transparent URL and form parameters

- URL (web::param) and form variables (web::formvar) separated
- handle multiple parameters with the same name

multi-part formdata

transparent parsing of urlencoded and
multipart/formdata form data

request handler abstraction

commands are independent of underlying request
handler - same accessor to CGI vars in CGI case and
in mod_websh

encoding

conversion from and to HTML code, uri encoding and
decoding

message protocol

send and receive data in platform-independent format

4 weshell tutorial

4.1 hello, world

```
web::put "Hello, world !<br>"
```

[demo](#)

4.2 command dispatching

```
web::command default {
    web::put "<a href=\"[web::cmdurl de]\">Hello, world
!</a>"
}
```

```
web::command de {
    web::put "<a href=\"[web::cmdurl \"\"]\">Hallo, Welt
!</a>"
}
```

```
web::dispatch
```

[demo](#)

4.3 logging

```
web::logfilter add *.alert-debug
web::logdest    add *.alert-error    command serious
web::logdest    add *.warning-debug  command normal
web::logdest    add -format {$f - $m} *.debug command
normal

proc serious {msg} {
    web::put "<font color=\"#ff0000\">$msg</font><br>"
}
proc normal {msg} {
    web::put "$msg<br>"
}
web::log demo.info "normal message"
web::log demo.alert "error message"
```

[demo](#)

4.4 sessions

```
web::cookiecontext mycookie
web::command default {
    mycookie::init cookiesample
    set cnt [mycookie::cget cnt 0]
    set newcnt $cnt
    incr newcnt
    mycookie::cset cnt $newcnt
    mycookie::commit
    web::put "looks like this is your visit Nr $cnt"
}
web::dispatch
```

[demo](#)

4.5 URL parameters and form variables

```

proc input {name size} {
    return "<input type=\"text\" name=\"\$name\"
value=\"[web::formvar $name]\" size=\"\$size\">"
}
proc submit {value} {
    return "<input type=\"submit\" value=\"\$value\">"
}
proc dl {code} {
    web::put "<dl>"
    uplevel $code
    web::put "</dl>"
}

proc dtdd {name} {
    foreach item [web::param $name] {
        web::put "<dt><b>$name</b></dt>\n<dd>"
        web::put "$item</dd>"
    }
}

proc form {action code} {
    web::put "<form method=\"POST\" action=\"\$action\"
enctype=\"multipart/form-data\">\n"
    uplevel $code
    web::put "</form>\n"
}

web::command default {
    dl {
        dtdd a
        dtdd b
        dtdd c
    }
    form [web::cmdurl "" [list a 10 a 20 b foo=bar c &]]
}

web::dispatch

```

[demo](#)

5 mod_websh

5.1 philosophy

reuse interpreters

do not spend time spawning a child or re-loading application logic

webshell script for everything

deploy *the same* webshell in CGI and mod_websh environments

template mode

if you must

Tcl is thread-safe - ideal for Apache 2.0

5.2 set-up

httpd.conf

```
LoadModule websh_module /path/to/mod_websh.so
WebshConfig /path/to/config_file.tcl
```

```
AddHandler websh .ws3
AddHandler websh .wsp
```

/path/to/config_file.tcl

```
web::interpclasscfg /path/to/my/shop.ws3 maxrequests 10
```

```
proc web::interpmap {f} {
    if {[string match *.wsp $f]} {
        return /path/to/my/wsphandler
    }
    return $f
}
```

/path/to/my/wsphandler

```
web::putxfile [web::request SCRIPT_FILENAME]
```

5.3 embedded execution

```
web::initializer {
  web::put "<tt>initializing code...</tt><br>"
  # open DB handle
  web::command default {
    web::put "<tt>"
    web::put "max requests allowed for this
interpreter:&nbsp;"
    web::put "[web::interpclasscfg [web::interpcfg]
maxrequests]<br>"
    web::put "current request handled by this
interpreter:&nbsp;"
    web::put "[web::interpcfg numrequests]<br>"
    web::put "</tt>"
  }
}
web::finalizer {
  # close DB handle
}
web::dispatch
demo \(one httpd\) demo
```

5.4 template mode

```
<html>
<head>
<title>&quot;Webshell Server Pages&quot;</title>
<link href="/samples.css" rel="styleSheet"
type="text/css">
</head>
<body bgcolor="#ffffff">

<h1>Welcome to &quot;Webshell Server Pages&quot;</h1>

Local time is <%web::put [clock format [clock
seconds]]%><br>

</body>
</html>
```

[demo](#)

5.5 architecture

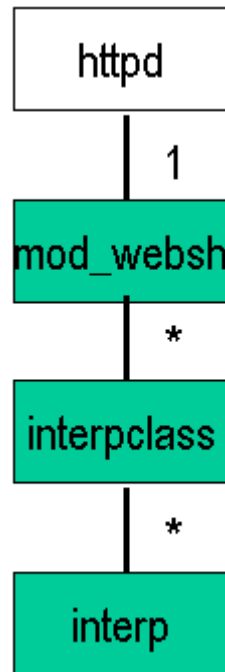


Figure 6 - Object Model

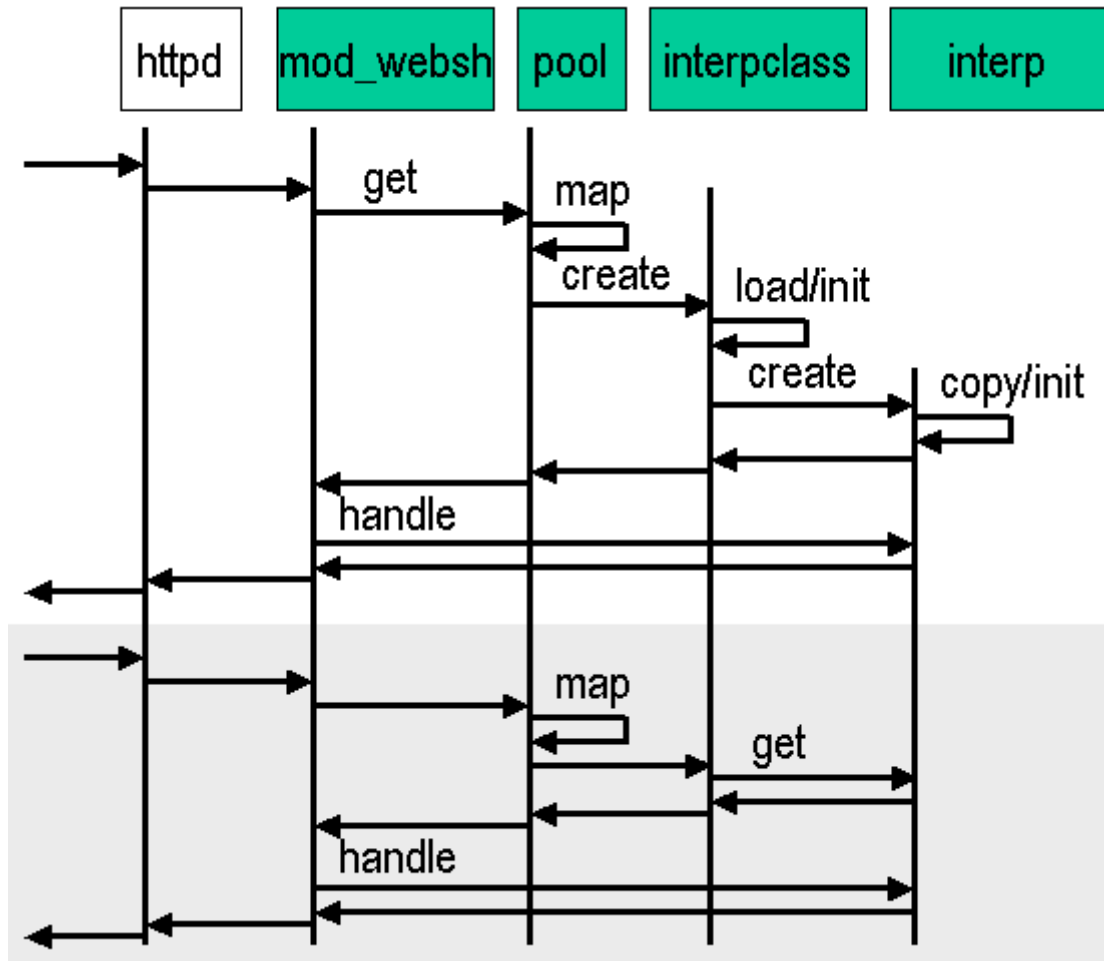


Figure 7 - Life Time of Interpreters

6 summary

6.1 webshell

web application development framework
reduced to the max

6.2 webshell is cool because

it covers all aspects of web applications

CGI, web server extension, templates

it has unique concepts

command dispatching

multiple output buffers

paradigm of one language for all

Tcl and webshell are thread safe

therefore, webshell is ready for Apache 2.0

it is Open Source Software

you have the code



Tcl/Web

- Questions?
- Comments?

VII
Q&A

39 | netcetera



Jochen Löwer (mailto:jochen_loewer@hp.com)

What is tDOM - (new) Features



- Tcl package for Tcl 8.x
- two XML parsers (Expat (1.95.1) + SimpleParser)
- **enhanced TclExpat (SAX)** (originally from Steve Ball / Scriptics)
- **DOM** implementation (DOM core level 1 + extensions)
- [incr Tcl] / **OO - like** calling syntax
- fast **XPath** implementation
- **high performance** (written in **C**)
- low memory consumption (**enhancements**)
- extension namespaces for DOM methods / XPath functions
- free for any use: MPL
- **partial XSLT support**
- **extension system and validation extension**
- **HTML parser and element builder**
- **more DTD information**
- **thread safeness**

Current version: tDOM-0.52

tDOM-0.6x in some weeks

tDOM, Jochen Loewer

13-Jun-01
1

Developers / Contributors



Jochen Loewer	Core (DOM, Tcl binding, Xpath, XSLT)
Rolf Ade	Extension Architecture, Validator, TclExpat enhancements,
Zoran Vasiljevic	Thread Support, Node commands (Tcl Dynamic Pages)

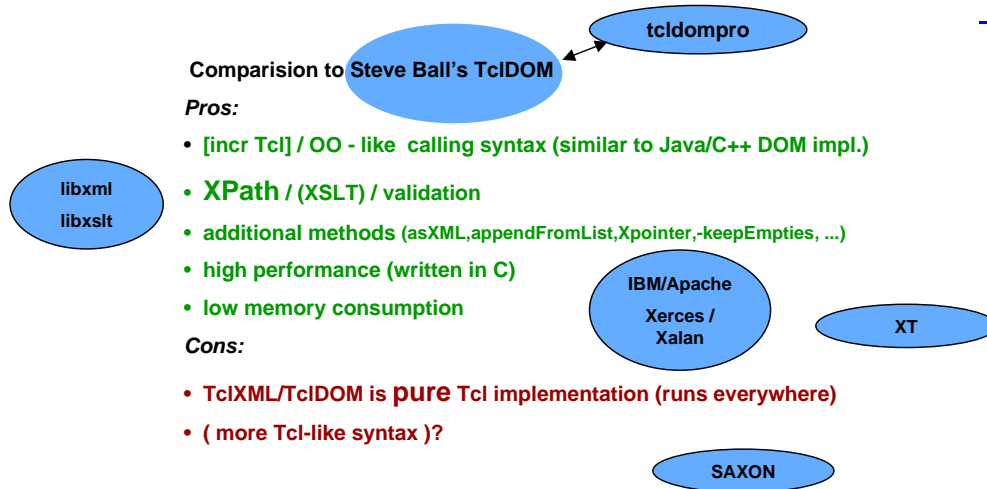
+ various bug reporters

- email based development work
- major releases done by J.Loewer
- use SourceForge for future ? (tdom.sourceforge.net already created, but not used)
- old major site (<http://sdf.lonestar.org/~loewerj/tdom.cgi>) is currently down!
- Maillinglist on www.egroups.com/group/tdom

tDOM, Jochen Loewer

13-Jun-01
2

Related Work - Pros / Cons



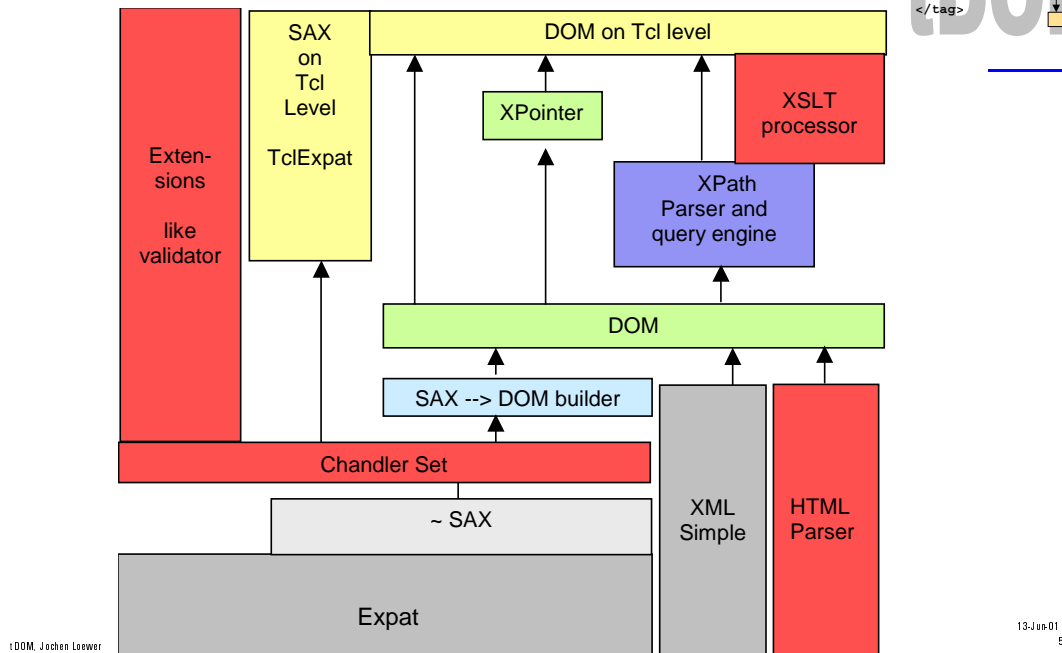
tDOM's usage in the world

- UML modelling tool (XMI) by a danish company
- BMEcat application (Rolf Ade)
- AOLserver modules (Tcl → DOM → HTML = Tcl Dynamic Pages) Zoran Vasiljevic
- configuration the high-end server (SuperDome)
- external system communication (logistic information HTTP/XML)
- frontend to backend communication
- ? C part used in a WML system/application in Hongkong ?

...

and lots of downloads from various organizations (IBM, Compaq, SUN, Software AG,....)

Architecture



OO-like Syntax / Object Commands

tDOM creates Tcl commands on the fly while traversing the DOM tree, which point to domNode C structures using their clientData:

`domDoc<N>` for DOM document objects
`domNode<N>` for all nodes (element, text, comment, PI)

No attribute, NamedNodeMap or DocumentFragments objects !

Basic syntax is:

`$obj method arg1 arg2 ...`

domNode2 command has not been created yet

return reference to domNode2 will create a command

```
% set doc [dom parse $xml]
domDoc1

% set rootNode [$doc documentElement]
domNode1

% domNode2 nodeType
invalid command name "domNode2"

% set child [$rootNode firstChild]
domNode2

% domNode2 nodeType
ELEMENT_NODE
```

Available DOM Methods for Nodes

domNode<n> **method** arg1 arg2 ...

basic properties

nodeType
nodeName
nodeValue
ownerDocument

navigate

parentNode
firstChild
lastChild
nextSibling
previousSibling
hasChildNodes
childNodes
getElementsByTagName

handle attribute

get/setAttribute
removeAttribute
hasAttribute
attributes
@<attr>

modify

appendChild
insertBefore
replaceChild
removeChild
cloneNode

Pls/text

target
data
text

Xpointer97 navigation search methods

find
child
descendant
ancestor
fsibling
psibling

XPath XSLT

selectNodes
xslt

serialize

asList
asXML
asHTML

add fragments

toXPath

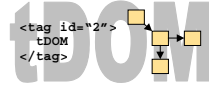
optional properties

appendFormList
appendFromScript
appendXML

getLine
getColumn

©DOM, Jochen Loewer

13-Jun-01
7



Parsing / DOM builder / Serialization

parsing / DOM creation

dom parse ?-ns? \$xml



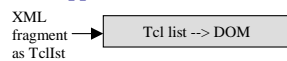
dom parse -simple \$xml



\$node appendXML

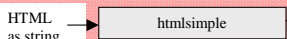


\$node appendFromList



\$node appendFromScript

dom parse -html \$html



serializing the DOM

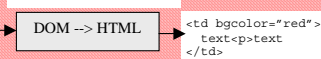
\$node asXML ?-indent 4?



\$node asList



\$node asHTML



©DOM, Jochen Loewer

13-Jun-01
8



New I/O for Parsers / Serializers



Instead of passing/retrieving a string, a Tcl channel or filename can be used

For XML parsing / DOM building:

```
$expatParserHandle parsechannel <ch>
$expatParserHandle parsefile <f>
$dom parse -channel <ch>
```

For serializers:

```
$node asXML -channel <ch>
$node asHTML -channel <ch>
```

Advantages:

- Avoids additional copy of data in memory
- Parse while data gets in and stop before the end

tDOM, Jochen Loewer

13-Jun-01
9

Namespace Support



While DOM building (dom parse) namespaces are parsed and stored in a per document table.

Nodes and attribute nodes contain 8 bit index to document namespace table (→ memory savings).

New methods:

```
$node namespaceURI
$node prefix
$localName
```

not (yet) implemented:

```
$node getAttributeNS
$node setAttributeNS
$node hasAttributeNS
$node getElementsByTagNameNS
```

Right now also the XML serializer doesn't create namespace definitions for elements/attributes (→ XSLT issue)

tDOM, Jochen Loewer

13-Jun-01
10

DTD Information



Moved back to standard Expat distribution (1.95.1 on SF) from hacked / improved version (by Scriptics/PerlXML/own hacks)

→ gives callbacks while DTD parsing (element / attribute declarations)

Input DTD:

```
<?xml version="1.0" ?>
<!DOCTYPE root [
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
]>
```

Callbacks on Tcl level:

```
XmlDecl 1.0 {} {}
StartDocTypeDel root NULL NULL 1
ElemDecl spec {
  SEQ {} {} {
    {NAME {} front {}}
    {NAME {} body {}}
    {NAME ? back {}}
  }
}
ElemDecl div1 {
  SEQ {} {} {
    {NAME {} head {}}
    {CHOICE * {} {
      {NAME {} p {}}
      {NAME {} list {}}
      {NAME {} note {}}
    }}
    {NAME * div2 {}}
  }
}
EndDocTypeDecl
```

could be useful for Validators +
Code Generators
(generate validate code,
generate object/data extraction code)

tDOM, Jochen Loewer

13-Jun-01
11

Thread Support



Initial modification for thread safeness did Zoran Vasiljevic for AOLserver integration last year.

Basically moved **global data** to **thread local storage**.

Each thread will have his own object counter, no conflicts, no locks needed,
but **no ability to share / hand over** DOM documents.

To enable this Zoran made a new implementation in May/June.

There will be **locks** on the whole document, which are controllable via two new methods.

Document objects will be passed between threads through object names containing the **physical address** (doc4f00340 → able to kill whole interpreter if bad address is used!).

There are currently discussion about a safer approach.

→ Thread support should have a compile time switch, so that non-threading tDOM uses gets no performance, robustness and complexity penalty!

tDOM, Jochen Loewer

13-Jun-01
12

Memory Consumption - DOM Allocator

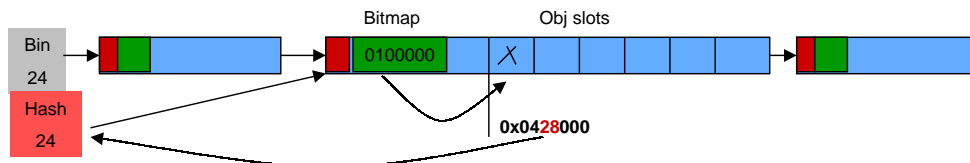
Many memory allocators can have quite large space requirements just for housekeeping information, usually around 8 bytes (linked list pointer + size)

DOM objects consists of many equally sized object → exploit this fact using a specialized allocator, which has a minimal overhead in these cases.

Idea:

- use large blocks (32K) for perfect fit of equal sized objects
- use bitmap vector for used/free tracking (→ 1 bit overhead)
- all blocks for object of that size go into one bin

-DUSE_NORMAL_ALLOCATOR
to disable it at compile time



Allocating is fine. Freeing gets complicated:

How to find the right block info structure just by the given memory address?

→ Use hashing of some middle address bits and comparing against begin/end address (cache line concept for 1st level CPU caches)

tdom, Jochen Loewer

13-Jun-01
13

xmlbench Performance Results

xmlbench suite (www.sosnoski.com/opensrc/xmlbench/results.html)

announced on www.xmlhack.com recently.

Test results made by Rolf this week on Win2000 (PII 333Mhz):

-Runtime(ms)-	much_ado	periodic	xml	
tdom SAX	771	591		
tdom DOM	1382	1101		
tdom DOM-simple	731	440		SAX: 3 times faster
Java SAX	2473	2153		
Crimson DOM	13149	9294		DOM: >9 times faster
JDOM	19458	10895		
dom4j	11557	8292		
Xerces DOM base	14361	10856		
Xerces DOM def	6429	5037		
Electric XML	17886	10095		

- Memory -	much_ado	periodic	xml	
tdom	332	212	284	
Crimson DOM	1101	603	817	3-4 times more momery
JDOM	1545	761	1025	
dom4j	1454	955	1167	
Xerces DOM base	1216	685	903	
Xerces DOM def	1065	738	1188	
Electric XML	1367	745	1089	

tdom, Jochen Loewer

13-Jun-01
14

HTML parser



- based on simple XML parser
- modifications to parse HTML <= version 4.0 code
- No double quotes / ticks for attribute values
- Get script / style tag content unparsed
- Be able to deal with empty tags

<p>
 <hr> ...

Main challenge:

To be able to deal with HTML coding errors !

```
<table> <tr> <td> Row1 Col1
          <td> Row1 Col2
        <tr> <td> Row1 Col1
          <td> Row2 Col2
```

```
</table>
```

```
<a href=/app.cgi?param=11><font color=white> LINK </a></font></a>
```

Idea: list of fields which could be autoclosed (under certain conditions)

ignoring some closing tags, which are left over

Heuristics need improvements!

Other sites won't change just because you can't parse them.

Advantage:

operate on the HTML document using DOM methods,
apply XPath queries

future: HTML -> DOM -> XSLT/Tcl -> WML ?

-> DoComo(?) HTML

(Japanese mobile HTML)

Usage of HTML parser



- HTML code analysis (browse for example with XE)
- HTML code condenser (rules to strip non visible white space, then asHTML)
- wrapper to (legacy) web application
- web robots / agents
- web service interfacing (-> WIDL WebMethods)

-> XPath features enable easy powerful scripting in Tcl !

Example: monitor / extract offering from Ebay

Web Extractor (Ebay)

Information to extract

WebCam Philips Vesta Pro!!! Neuwertig
 Artikelnummer 1243380681

Kategorie: Computer & Computerspiele Hardware Multimedia Webcams

Aktuelles Gebot: **90,00 DM**

Startpreis: **90,00 DM**

Menge: 1

Verbleibende Zeit: **6 Tage(n), 4 Stunde(n) +**

Start: 03.06.01 21:25:15 MESZ

Endet: 10.06.01 21:25:15 MESZ

Verkäufer (Bewertung): **houhock (1)**

Höchstbieter: --

Zahlung: Per Nachnahme, Überweisung, Barzahlung bei Übergabe. Siehe Artikelbeschreibung

Versand: Käufer trägt die tatsächlich anfallenden Versandkosten

Artikel aktualisiert: Verkäufer: Wenn für diesen Artikel noch keine G...

Verkäufer trägt die volle Verantwortung für das Anbieten dieses Artikel
 Fragen zu klären.

Neue WebCam Philips VestaPro PCVC680K (einmal gebraucht) mit U-
 Stativ für Windows und Mac!!! Videoclips in VGA Qualität mit 30 Bild-
 Auflösung (800*600) Inklusive zwei CDs mit echt guter Software (Tre-
 2 SE, Media Studio Pro und NetMeeting 2.1) Verschicken Sie doch ein
 echtes!!! Käufer trägt Versandkosten!

Preis	Anzahl	Zeitpunkt
76,00 DM	3	08. Jun. 19:30
136,00 DM	3	09. Jun. 13:46
90,00 DM	-	10. Jun. 21:25

13-Jun-01 17

XPath Queries: Example

all descendant element nodes

Predicate / Subfilter

`//country[name='Germany']/province[population > 5000000]/name`

filter all elements with nodeName 'country'

get all child nodes

implicit convert of text nodes below 'population' tag into numbers

```

<country>
  <name>Germany</name>
  <province>
    <name>Berlin</name>
    <population>3472009</population>
  </province>
  ...
  <province>
    <name>Bayern</name>
    <population>11921944</population>
  </province>
  ...
</country>
  
```

'Join' over reference id values out of different parts in the XML:

`//mountain[in_country[@ref = string(//country[name='Germany']/@id)]`

13-Jun-01 18

Web Extractor (Ebay) – XE examples 2



```
xml(file:/home/jolo/ebay2.html) /
- html
+ head
C header for browse: view item
C begin header
- body BGCOLOR=#FFFFFF
+ table BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=600
+ center
+ center
+ center
= br
+ table border=0 cellpadding=8 cellspacing=0 width=100%
+ center
- blockquote
T
  Neue WebCam Philips VestaPro PCVC680K (einmal gebraucht)mit USE
  Käufer trägt Versandkosten!
+ a name=ebayphotohosting
```

```
xml(file:/home/jolo/ebay2.html) string( /*
                                     [contains(., "Aktuelles Gebot")]
                                     /following-sibling::td[1]
                                     )
90,00 DM
```

tDOM, Jochen Loewer

13-Jun-01
21

Handler Sets for Expat / Stacked tDOM

developed by Rolf Ade



For DOM building the Expat callbacks (SAX events) are tight to the DOM object creation functions.

Why not having having the SAX events trigger other actions (element statistics, validation, ...) beside the standard DOM builder in parallel?

Having a subsequent parse to accomplish that is not a very clever alternative!

Idea:

Extent Expat to be able to invoke multiple callbacks for an event → **CHandlerSets**

CHandlerSets provide a stackable modular extension mechanism, which allows to write extensions separate from the main tDOM distribution



tDOM, Jochen Loewer

13-Jun-01
22

tnc Extension: DTD based validation

developed by Rolf Ade



The tnc extension package uses the CHandlerSets and allows fast C speed DTD based validation while building up the DOM tree at the same time.

```
package require tdom
package require tnc

roc LoadAndValidate { xml } {
    set parser [expat]
    tnc $parser enable
    tdom $parser enable
    $parser parse $xml
    set doc [tdom $parser getdoc]
    puts [[ $doc documentElement] asXML]
    $parser free
    return $doc
}

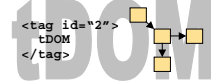
LoadAndValidate {<?xml version="1.0"?>
    <!DOCTYPE Test [ <!ELEMENT Test (#PCDATA) > ]><Test></Test>}
LoadAndValidate {<?xml version="1.0"?>
    <!DOCTYPE Test [ <!ELEMENT Test (#PCDATA) > ]><TestFoo></TestFoo>}
```

tdom, Jochen Loewer

13-Jun-01
23

NodeCmd

developed by Zoran Vasiljevic



Fast C implementation for DOM node creation, which could nest.

Example:

```
% dom createNodeCmd elementNode html::body
% dom createNodeCmd elementNode html::title
% dom createNodeCmd textNode      html::t
```

And usage:

```
% set d [dom createDocument html]
% set n [$d documentElement]
% $n appendFromScript {
    html::body {
        html::title { html::t "This is an example" }
    }
}
% puts [$n asHTML]
<html>
  <body>
    <title>This is an example</title>
  </body>
</html>
```

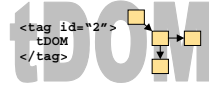
This is the foundation for Zoran tdomtdp package for AOLserver: **Tcl Dynamic Pages**

tdom, Jochen Loewer

13-Jun-01
24

Tcl Dynamic Pages - Examples

developed by Zoran Vasiljevic



Example:

```
body {
#
# Server info
#
h2 {t "Server Information"}
blockquote {
table {
foreach item {
server hostname address pid uptime boottime home config log
pageroot tcllib nsd argv0 name version label builddate platform
} {
tr {
td -align right - valign top {b {t "$item: "}}
set itemval [ns_info $item]
if {$item == "boottime"} {
set itemval [ns_httpstime $itemval]
}
td -align left - valign top {t "$itemval\&nbsp;" }
}
}
}
}
```

similar to Don Libes' `cgi.tcl` approach, but builds DOM tree fragments (in C).

At the end the DOM tree is serialized with `[$root asHTML]`

tDOM, Jochen Loewer

13-Jun-01
25

XSLT processor (in C)



Right now there is no directly Tcl embedded XSLT engine available (Steve Ball did some wrapper to XT/SAXON/... outside as a process)

Implementation started end of July 2000, most code done until October.

Got final approval to also release it as Open Source in December.

Currently ~ 70 KByte C code and a new code in domxpath (LocationPath matcher)
(obj code: 55K for XPATH, 26K for XSLT)

Problem: A lot of templates have to be checked in parallel

```
<xsl:template match="book/author">
...
</xsl:template>
<xsl:template match="book/title">
...
</xsl:template>
```

Eval XPathExpr

AxisChild book

AxisChild author

LocPathMatches(node)

IsElement author

ToParent

IsElement book

match contains LocationPath and not XPath expressions. So current XPath implementation doesn't help much and evaluates for the current node downwards.

- new LocationPath parser had to be coded
- bottom-up match approach

tDOM, Jochen Loewer

13-Jun-01
26

XSLT processor (in C) – Usage



```
$root xslt $xsltDoc transformedDoc  <-- change to document method

proc ApplyTemplate { xml xslt } {
    dom parse -keepEmpties $xml  xmlDoc
    dom parse -keepEmpties $xslt xsltDoc
    [$xmlDoc documentElement] xslt $xsltDoc transformedDoc
    # depending on the output type
    [$transformedDoc documentElement] asXML
    [$transformedDoc documentElement] asHTML
}
```

tDOM, Jochen Loewer

13-Jun-01
27

XSLT processor (in C) – Current State



Other work / competitors:

C: libxml/libxslt from D.Veillard (GNOME)
 C++: Sabletron
 Java: XT James Clark
 Saxon
 Xalan (IBM -> Apache XML project)

currently already passed:

Mozilla XSLT engine tests
 a great part of tests from LibXML/LibXSLT (GNOME, D. Veillard)
 some other XSLT test from various sources
 some of M.Kay's tests
 Joe English TMML Tcl man page formatter does not work completely (key/import is missing)

current problems:

- document fragments for xslt:variables / XPath expression
- xslt:number, xslt:sort, ... partial implemented
- function format-number (Tcl based implementation initially preferred)
- missing:
 - element creation with namespaces
 - output type determination and handling (only text, may need asText)
 - xslt:key
 - xslt:import / xslt:include / xslt:apply-import

→ not usable in production code, if all features are needed!

tDOM, Jochen Loewer

13-Jun-01
28

XPath / XSLT extension functions in Tcl



If function is not found in XPath / XSLT processing, a callback can try look for a Tcl-level implementation:

```
proc ::dom::xpathFunc::format-number { ctxNode pos nodeListType nodeList args } {
    set argLen [llength $args]
    if { ($argLen != 4) && ($argLen != 6) } {
        return -code error "wrong number of args: format-number(node,typeString,?decFormat?)"
    }
    foreach { arg1Typ arg1Value arg2Typ arg2Value } $args break
    set num [::dom::xpathFuncHelper::coerce2number $arg1Typ $arg1Value]
    set formatStr [::dom::xpathFuncHelper::coerce2string $arg2Typ $arg2Value]
    ...
    return [list string $num]
}
```

Good way to get something implemented and working first on Tcl level.
Later recode in C can be made, if performance matters.

Future / Enhancements



- finalize thread support
- xslt bugs fixes and enhancements
- namespace enhancement (c level creation functions, tcl level methods, namespace support in XML serializer)
- release tDOM-0.6x soon.
- PURL for tDOM / sourceforge (?)
- XML schema validation
- SOAP client/server
- UDDI ?
- new XML Query proposal by W3C (Software AG,...)

Poll:

What do want to have?

Use of XML or XML-based RPC technics (SOAP) in future projects?

Thanks



- Thanks for interest
- Call for help / volunteers / testers / users.
- Questions?



Game Scripting with Tcl

Carsten Orthbandt (<mailto:carsten.orthbandt@sek-ost.de>)

Director Development

SEK-Ost (<http://www.sek-ost.de>)

Game Scripting with Tcl

Carsten Orthbandt
 Director Development
 SEK-Ost
<mailto:carsten.orthbandt@sek-ost.de>
<http://www.sek-ost.de>

SEK.Ost is a German game development company. For our current project “Wiggles”, we decided to employ a scripting system to make content creation easier and faster. Going with Tcl was a key factor in realizing our concept.

1 Wiggles

Wiggles is a 3D realtime strategy game with strong emphasis on character development. You command a bunch of gnomes, each having its own mind and abilities. Though the game starts at the surface you quickly begin to dig tunnels and caves. While the player has direct control over each of the gnomes, they are able to perform most tasks automatically. Wiggles runs on Microsoft Windows computers with DirectX.



2 Scripting game content

The complexity of current computer games enforces new techniques for content creation. While most games use core engines written in C++, level design and game logic are usually implemented with scripting languages and specialized design tools. Graphical level editors are great tools for manipulating objects in your game world but defining complex behaviours through a GUI makes maintenance a very time consuming tasks. This is where scripting languages come in and save the day. Game development is a rather chaotic process where standard methods of software engineering are only partly adopted or simply do not work at all. There are many scripting languages in use today and developers are constantly reinventing the wheel.

2.1 Requirements for games

The ideal language for game content creation has to meet several goals at once.

2.1.1 High performance

With constantly rising levels of graphics detail and game depth, simulation and game logic must be fast. Current games show hundreds or thousands of objects each being rendered and simulated. Memory consumption and execution speed are key factors.



2.1.2 Ease of use

Most game designers are not programmers, they don't need complex languages but productivity. Game content design requires a language that is easy to learn and teach. Error messages should be clear. Scripting must never cause protection faults or other system errors.



2.1.3 Expandability

Structures for interactive content often do not translate well into given language constructs. A scripting language should provide methods to extend its syntax. Most games have their own system abstraction layer that is not reflected by standard scripting languages. Therefore scripting integration with the game core systems should be lightweight and easy to implement.

3 Tcl the Wiggles way

At SEK.Ost, Tcl has proven to be a valuable tool not only for direct game control but also for workflow tools in the production process. We built a wrapper module to simplify Tcl access from C++. This wrapper is used everywhere in our code and has some special features to solve some real life problems we had.

All Tcl applications at our site are embedded Tcl. Nobody runs `telsh`. Every tool embeds and extends Tcl as a standalone application. Since we cannot rely on a Tcl installation on the customers computer we do not use Tcl DLLs but link statically.

3.1 Wiggles game objects

Every game object in Wiggles has a number of attributes like position, rotation, velocity and animation status. Simple behaviours are performed through actions, which are the basic building blocks for more complex tasks defined through Tcl scripts.

For every object there is a Tcl class that defines states, methods and event handlers. The core C++ engine provides commands for handling objects, classes and independent game data. Apart from that all game rules are defined by Tcl scripts.

3.2 Tcl_Obj wrapper class

The core of our wrapper are two C++ classes that encapsulate Tcl objects and interpreters. This is our CTclObj definition:

```
class CTclObj
{
    Tcl_Obj*      m_pO;
public:
    CTclObj();
    ~CTclObj();

    CTclObj(Tcl_Obj* pOrg);
    CTclObj(const CTclObj& org);
    explicit CTclObj(const char* pcText);
    explicit CTclObj(const int iValue);
    explicit CTclObj(const double dValue);
    explicit CTclObj(const CVec3D& vVector);
    explicit CTclObj(const bool bValue);

    CTclObj& operator=(const CTclObj& org);
    CTclObj& operator=(Tcl_Obj* pOrg);
    bool operator==(Tcl_Obj* pOrg) const;
    operator Tcl_Obj*() const;

    int GetInt() const;
    double GetDouble() const;
    float GetFloat() const;
    const char* GetString() const;
    CVec3D GetVector() const;
    bool GetBool() const;

    bool CvtToInt(int& i) const;
    bool CvtToDouble(double& d) const;
    bool CvtToFloat(float& f) const;
    bool CvtToVector(CVec3D& v) const;
    bool CvtToBool(bool& b) const;

    bool ListAdd(const CTclObj& toElem);
    int ListLength();
    CTclObj ListGet(int iIndex);
    CTclObj ListRemove(int iIndex);
    CTclObj ListReplace(int iIndex, CTclObj toNew);
    static CTclObj ListCreate();

    CTclObj Duplicate();

    static CTclObj FromFile(const char* pcFilename, bool bStripCPPComments=true);
    static CTclObj FromFileNoCache(const char* pcFilename, bool bStripCPPComments=true);

    static void EnableCache();
    static void DisableCache();
    static void FlushCache();
    static void GetCacheRates(int& iHits, int& iMisses);
    void DoAutoArc(AutoArc& arc);

    static void PreProcReset();
    static void PreProcDefine(const char* pcDefine);
    static void PreProcUndefine(const char* pcDefine);
    static bool PreProcIsDefined(const char* pcDefine);
};
```

Most of these methods translate forward to the Tcl API, but there are some notable extensions to the usual Tcl functionality.

3.2.1 Pre-processing

There is no doubt the normal Tcl comment (#) is not strictly intuitive to use. After some entertaining problems with the hash comment, we decided to provide a different mechanism for comments. We simply copied the C++ line comment (//):

```
proc test {} {
    for {set i 0} {$i<10} {incr i} {
        // this comment works ! {
        puts $i // this one does not
    }
}
```

This new comment is only valid at the start of a line, ignoring whitespace. Obviously this requires some pre-processing for all scripts. This is accomplished through the `FromFile*` methods. These methods replace the Tcl `source` command functionality. We decided not to redefine any Tcl core commands and provide a new command `"call"` that goes through this pre-processing.

When developing a game, you have to produce demo versions frequently. For space and security reasons, these demos have to be crippled. Since maintaining content for two different versions (demo and full) consistently is next to impossible, we extended our pre-processor further to allow `#define`-like parsing.

```
//# IF FULL
set info "[lmsg Language] Full version"
//# ELSE
set info "[lmsg Language] Demo version"
//# ENDIF
```

Though this is neither very elegant or “Tclish”, it has proven to be a powerful feature. The full set of pre-processor commands is show below:

```
//#define <id>
    Sets the switch <id>.
//#undef <id>
    Unsets switch <id>.
//#stopif <id>
    Stops parsing of file if <id> is defined.
//#stopifnot <id>
    Stops parsing of file if <id> is not defined.
//#if <id>
//#ifnot <id>
//#else
//#endif
    C-like conditional inclusion of Tcl source. Can be nested.
```

When building a demo version we strip the unwanted content by simply loading the scripts and saving them back to the demo location. The resulting demo script files do not contain any comments or pre-processor statements since these were filtered on loading.

3.2.2 Bytecode cache

Every created object in the game calls a class script, which in turn may call several other shared scripts. We found that loading and compiling these scripts on every object creation causes a significant performance hit. To circumvent reloading and compiling of otherwise constant scripts, we added a cache that keeps all loaded scripts in memory and returns the pre-processed and (if it was evaluated before) compiled Tcl object on successive requests for the same file. Although there are more than 250 script files for object class definitions in our game, the cache is very efficient. Typically the cache hit rate is well above 80% just after game startup and >99% after a few minutes.

The cache can be flushed through `CTclObj::FlushCache()`, or completely disabled.

3.3 Tcl_Interp wrapper class

```
class CTclInterp
{
    Tcl_Interp*    m_pI;
    bool m_bOwn;

    // when this use counter goes Zero, Tcl_Exit(0) is called
    static long g_iUseCounter;
    CTclInterp& operator=(const CTclInterp& ipOrg){return *this;};
    CTclInterp(const CTclInterp& ipOrg){};

public:
    CTclInterp();
    ~CTclInterp();

    // create an interpreter
    void Init();
    void Shut();

    bool Initialized() {return m_pI!=NULL;};
    operator Tcl_Interp*() const;

    void SetUserData(const KStr& sKey,ClientData pData);
    ClientData GetUserData(const KStr& sKey);

    CTclInterp(Tcl_Interp* pInterp);
    CTclInterp& operator=(Tcl_Interp* pInterp);

    KStr m_sError;
    KStr m_sErrorShort;

    bool SetVar(const CTclObj& sVarName,const CTclObj& sVarValue);
    CTclObj GetVar(const CTclObj& sVarName);
    bool UnsetVar(const KStr& sVarName);
    bool SetArrayVar(const CTclObj& sVarName,const CTclObj& sKey,
                    const CTclObj& sVarValue);
    CTclObj GetArrayVar(const CTclObj& sVarName,const CTclObj& sKey);
    bool UnsetArrayVar(const KStr& sVarName,const KStr& sKey);

    bool Eval(const CTclObj& oScript);
    CTclObj GetResult();
    CTclObj EvalExpr(const CTclObj& oExpr);
    bool EvalExprBool(const CTclObj& oExpr);
    bool EvalScope(const CTclObj& oScript);
    bool EvalScope(const CTclObj& oScript,CTclObj& toParamNames,CTclObj& toParamVals);

    void AddCmd(const KStr& sCmdName,Tcl_ObjCmdProc* pProc,ClientData cd);
    void AddCmdList(const CTclCmdList& cmdList,ClientData cd);
    void RemCmd(const KStr& sCmdName);

    void SetResult(CTclObj& toResult);
    void SetError(const char* pcReason,const CTclObj toOffendingObj=0);
    void SetTypeError(const CTclObj toOffendingObj=0);
    void SetArgCountError(int iNeededArgs);

    bool DefineProc(const char* pcProcName,const CTclObj& toArgs,const CTclObj& toBody);
    bool ProcExists(const char* pcProcName);
    bool CallProc(const char* pcProcName,CTclObj& toArgs);
};
```

Again, most of this class mirrors the Tcl API closely. We added extended error information that simplifies error location in script files. To simplify creation of special language structures, we also added methods for handling procs from the C++ level.

3.4 Tcl usage in Wiggles

3.4.1 Class properties

There are many object types with differing complexity. These range from simple decoration objects to the gnomes with dozens of event handlers, methods and attributes.

Object behaviour in Wiggles is defined through its “class”. When a new object is created, it creates an independent Tcl interpreter that runs object initialisation according to the class definition.

Object classes define:

- variable initialisation values
- public object methods
- event handlers
- state handlers
- appearance defaults

Because every active game object runs its own interpreter, there has to be a way to let objects communicate. This is done through methods and events.

Methods work like normal procedures, but can be called across interpreter boundaries.

Methods may return values, but most do not. When running a networked game, method calls may transform into remote procedure calls on some other computer.

This is a typical method, taken from the “Troll” class.

```
method burn {} {
    if { $burning } { return }
    set burning 1
    add_attrib this atr_Hitpoints -1.2
    action this anim burn {state_enable this} {state_enable this}
    change_particlesource this 0 27 {0 0 0} {0 0 0} 256 16 0 0 0 1
    set_particlesource this 0 1
}
```



Events can be fired by the core C++ engine or other objects. We define a global list of event types that each object may handle. If a class should handle some event, it defines a handler:

```
handle_event evt_task_defend {
    tasklist_clear this
    set attack_item [event_get this -subject1]
    set attack_behaviour "offensive"
    set approach 0
    fight_startfight
}
```

States are a simplified form of finite state machines. Each object has one active state. State processing can be disabled and enabled. A common pattern is:

- receive event
- start action
- disable current state
- core engine executes action
- action finishes
- re-enable object state

Actions are basic behaviours provided by the core C++ engine. These actions include animation, walking and simple sleeping. When starting an action, the script specifies two Tcl snippets that are executed when the action finishes. There is one “finish handler” for successive action termination and one handler for termination due to errors:

```
action this anim burn {state_enable this} {state_enable this}
```

All commands on objects take an object reference as the first parameter. These references may be a simple ID of an object or the special “this” reference for the object that owns the interpreter the script is running in.

3.4.2 Custom commands

There are over 600 custom commands in Wiggles, 300 of them being object bound. As an example I’ll describe the `set_attr` command:

```
//@TCL set_attr <objref> <attribname> <value>@attrib@ set new attribute value
int CTclObjMethods::SetAttrib(ClientData cd,Tcl_Interp* pI,int iObjc,Tcl_Obj* const vObj_[])
{
    TCLTRACE("CTclObjMethods::SetAttrib()");
    CTclObjInterp ip(pI);
    CTclVarArgs vObj(iObjc,vObj_);
    CObjPtr pObj=GetObjSafe(pI,iObjc,vObj_,4);
    if(!pObj){return TCL_WARNING;};
    CTclObj toAttr(vObj[2]);
    CTclObj toValue(vObj[3]);
    pObj->m_Attrib.Set(toAttr.GetString(),(float)toValue.GetFloat());
    ip.SetResult(CTclObj(pObj->m_Attrib.Get(toAttr.GetString())));
    return TCL_OK;
};
```

The first line is a special comment for documentation. We built a tool that scans through the C++ sources and creates HTML documentation from these comments.

`TCLTRACE()` is a macro that enables performance profiling of Tcl custom commands in debug builds. We use this information to find hot spots and time critical functions.

Each custom command uses the given interpreter through a wrapper (`CTclObjInterp`) that provides enhanced evaluation methods for profiling and error tracking.

`CTclVarArgs` wraps the arguments in a safe array that is bounds checked in debug builds. The object reference in parameter 1 is converted to a pointer and checked for validity. This conversion also checks the parameter count and sets the interpreter error if it does not match. The other parameters are then used to actually set the attribute value and the interpreter return object.

3.4.3 Performance considerations

The memory overhead of having one interpreter per game object is fairly high (~35 kByte). With thousands of game objects we had to find a way to reduce memory usage.

Since most object are for decoration, they do not have any handlers or methods. The runtime detects classes that only need object initialisation and creates these objects without an interpreter using the defaults given on class definition. These “dumb” objects can still be accessed from other (active) objects through their Tcl commands, but may not execute any Tcl code.

Each execution of custom commands, states, events or methods is tracked for statistics. These statistics are dumped to a text file after program termination. From this we can find slow Tcl code or custom commands that are good candidates for optimisation.

Most non-trivial handlers have their real code moved to procs for better bytecode optimisation.

3.5 Localisation

All user-visible text is held in language-dependent files and retrieved through a custom command “`lmsg`” that returns the translated version from a hash table. This technique is also used for texts in the core engine.

3.6 In-game shell and web server

For testing and development we included a simple shell in the game. Every action in the game can be triggered through commands in the shell. This shell is also accessible through an integrated web server for remote troubleshooting. These two tools provide lots of information on game state and means to perform actions that are not possible with the official user interface.

3.7 Limitations

Wiggles does not support console I/O because there is no shell window. There is no support for packages or extensions not provided by the hosting application. There is no trace of the standard Tcl distribution in our game. Therefore much Tcl standard stuff does not work in our game.

4 Things we didn't do

There are a few things we could have done but decided against. Some simply didn't work, others were too slow.

4.1 [incr Tcl] OOP

Although there is an object class concept in Wiggles, there is no inheritance. This reduces the requirements for a Tcl OOP system drastically. When we had a look at [incr Tcl], we found it quite powerful but also far more advanced than “naked” Tcl. There was no convenient way of interaction between the core C++ engine and iTcl defined class objects. It seems not to be designed to be embedded.



4.2 Custom object types

Handling C++ object references in Tcl was one of the bigger challenges. Objects might be deleted while a script still has a reference. We considered building a custom type for smart object pointers, but it did not work out. We would have needed much more control over lifetime of custom type internal representations.

4.3 Tcl scripted user interface

The user interface of Wiggles is completely written in C++. We tried to keep the interface very simple. Although Tk shows that Tcl works well for interfaces, computer games tend to constantly break the rules of interface design. Therefore we'd had to implement completely new UI functionality for Tcl.

4.4 SWIG wrapper generation

SWIG does a great job of generating Tcl integration for C code. But most C++ methods of the core engine are rather low-level and not suited for Tcl commands. The `set_attr` command shown above is the probably most simple function in Wiggles. The majority of custom commands uses sequences of C++ methods to perform higher-level tasks. Therefore

automatic wrapper generation did not help with the functions that actually take the most time to write.

5 Things we might do on our next project

Wiggles was our first big-scale use of Tcl and we gained much experience. Naturally, there are many things we would do very differently if started the project today. For our next project, we are considering some changes.

5.1 Lightweight Tcl

The Tcl core is very efficient but also very big. We do not use events, channels, sockets or regular expressions. We'd like to remove these and other features from the Tcl core to make it less memory hungry. This showed to be anything but trivial. After removing all core commands we do not need, the library was only 16 kBytes smaller. Some of those features are very tightly integrated with the interpreter. There is much room for improvement.

5.2 Reimplementing Tcl

It might be simpler to do a complete rewrite of Tcl than to strip the unwanted features. There are things we'd like to do totally different, among them being the management of custom types (see 4.2) and the possibility of opaque types.

A complete rewrite would take much time and will probably be less efficient. But it seems very difficult to customize the existing core to fit our needs.

5.3 Tcl build tools

We are considering using Tcl as a sort of “make” tool. The tmk project is very promising but far from being usable on Windows platforms. It might be practical to implement the subset we would use.

5.4 Binary scripts

For performance and security reasons we'd like to store the game scripts in compiled binary format. Wiggles stores scripts as encrypted text files.

6 The end

Scripting as means of content creation is widely used in the computer games industry. Being only one of many current approaches, Tcl showed to be just the right thing for Wiggles.

7 References &

Wiggles Homepage

www.wiggles.de

www.wiggles.org

Links





Generating test programs with TestMake

Arjen Markus (<mailto:Arjen.Markus@wldelft.nl>)
1 WL | Delft Hydraulics

Generating test programs with TestMake

Arjen Markus¹
WL | Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

Abstract

Three aspects of Tcl make it a very suitable scripting language for generating (test) programs in, say, Java or FORTRAN 90: its abilities to manipulate long strings or text fragments, its support for associative arrays and the ease with which data can be interpreted as Tcl code. This way, parsing input data is almost trivial.

The application described here exploits these features to generate test programs. The user identifies a piece of code, one or several subroutines for instance or the source code for a whole class that need testing in some isolated environment. The input, as far as the user is concerned, mainly consists of:

- The declarations of input or output variables, possibly with a suitable test to see if the code under test does its job.
- A set of test cases which exercise the code.

It is then the task for the application to generate a complete program from these specifications.

The advantages of this approach are that one is concerned only with the formulation of the test cases (with the help of static analysers even that may be automated to a certain extent) and the evaluation of the results. The details of the program that should run all these tests are taken care of by the general Tcl script.

Introduction

This paper describes a Tcl application that builds (test) programs from straightforward specifications. The idea for creating an application that would generate a complete test program based on some fragments of code developed slowly:

- First of all, Tcl comes with an extensive facility to build and execute test suites. This, however, requires one to specify the outcome of a test as a single string, whereas the results of a (FORTRAN) routine might be an array of real values that can be checked against some criterion.
- Second, building test programs is repetitious and tedious. It means, implementing a series of tests where you have to specify all code to check the result and report about it as well as making sure that all tests are run in the correct way.
- Third, some analysis tools are capable of reporting the conditions by which the flow of control would follow a certain path through the code (see figure 1; ref 1.). This is valuable information when examining the code interactively, but it could be used for generating test cases as well.

¹ E-mail address: arjen.markus@wldelft.nl

- The fourth and last source of inspiration was a paper on testing the implementation of POSIX routines on a variety of machines (ref. 2.) In this work, fragments of code were used to construct an almost exhaustive set of test programs by which the various error conditions of a POSIX routine could be exercised.

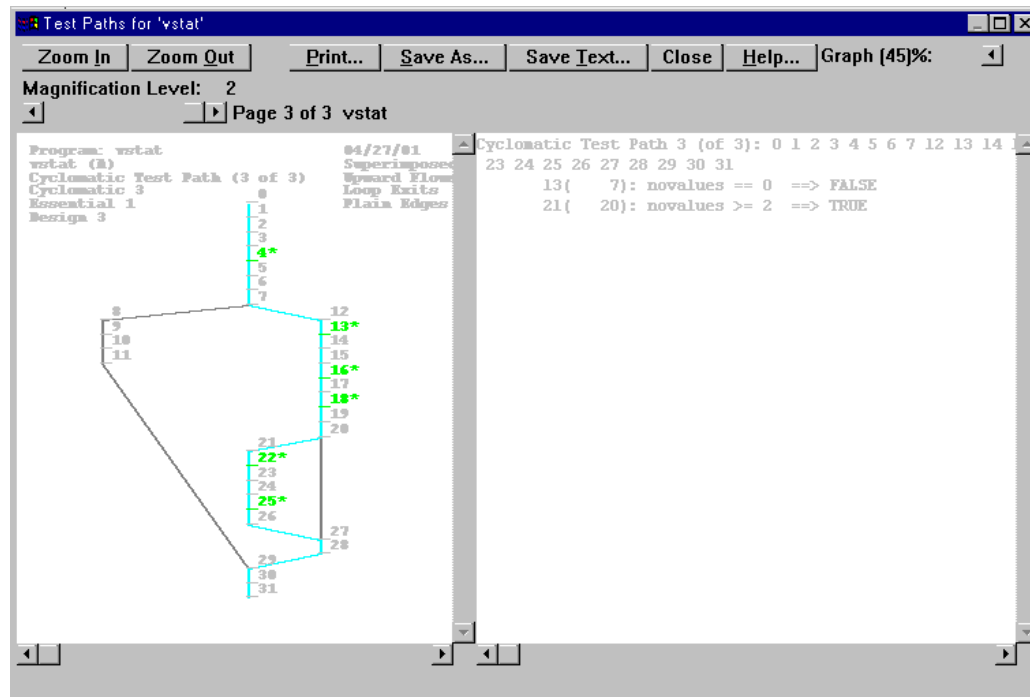


Figure 1. Example of a test path, as generated by a static analysis tool.

A typical situation

To make the idea less abstract, let us examine an example more closely, a hypothetical program that analyses measurement data. Such a program, written in the language of your choice, will have to perform a number of tasks like:

- Gather the input data
- Do the actual analysis
- Report the results

Hence, a coherent part of the program could be a single routine or a set of routines that determine simple statistical parameters (the mean, the extremes, the standard deviation). Such coherent parts can be tested more or less in isolation and that is what TestMake tries to facilitate. The routine below in FORTRAN 90 is one implementation²:

```
subroutine vstat( values, vmean, vmin, vmax, vstd )
  implicit none
  real, dimension(:), intent(in) :: values
  real, intent(out) :: vmean, vmin, vmax, vstd

  integer :: i
  integer :: novalues
  real, parameter :: vmiss = -999.0
```

² For the sake of brevity all comments have been stripped. There is also no claim to good programming practice.

```

novalues = size(values)

vstd = vmiss
if ( novalues == 0 ) then
  vmean = vmiss
  vmin = vmiss
  vmax = vmiss
else
  vmean = sum(values) / novalues
  vmin = minval(values)
  vmax = maxval(values)
  if ( novalues >= 2 ) then
    vstd = sum(values**2) - vmean ** 2 * novalues
    vstd = sqrt( vstd / ( novalues - 1 ) )
  endif
endif

return
end subroutine vstat

```

The Java class that follows has the same functionality, although it requires quite a different call sequence:

```

import java.lang.* ;
import java.math.* ;

public class vstat {
  public final float missing_value = -999.0f ;
  private int    novalues ;
  private float vsum ;
  private float vsum2 ;
  private float vmin ;
  private float vmax ;

  public vstat() {
    restart();
  }

  public void add( float value ) {
    novalues ++ ;
    vsum += value ;
    vsum2 += value*value ;
    if ( value > vmax ) vmax = value ;
    if ( value < vmin ) vmin = value ;
  }

  public void restart() {
    novalues = 0 ;
    vsum = 0.0f ;
    vsum2 = 0.0f ;
    vmax = -Float.MAX_VALUE ;
    vmin = Float.MAX_VALUE ;
  }

  public float average() {
    return (novalues>0)? vsum / (float)novalues : missing_value ;
  }

  public float stdev() {
    float stdv ;
    if ( novalues > 1 ) {
      stdv = ( vsum2 - vsum * vsum / (float) novalues ) / (float) (novalues-1) ;
      stdv = (float) Math.sqrt( (double) stdv ) ;
    } else {
      stdv = missing_value ;
    }
    return stdv ;
  }

  public float min() {
    return (novalues>0)? vmin : missing_value ;
  }

  public float max() {
    return (novalues>0)? vmax : missing_value ;
  }
} // End of class

```

Inspection of the tasks these *modules* (the term used in TestMake, for lack of anything better and less worn out) perform, reveals a number of test criteria:

- The mean and extreme values can be ordered as: $\text{minimum} \leq \text{mean} \leq \text{maximum}$, unless there are no values.
- The standard deviation must be non-negative (otherwise there will be a domain error when the square root is evaluated).
- The statistical parameters can only be determined if there are enough values. Otherwise this must be marked by, say, a reserved value like -999.0.

The modules can be tested with test cases such as:

1. There are no data
2. There is only one value
3. There are two or more different values
4. There are only two measurement data with the same value (which means the standard deviation should be zero!)
5. Other test cases whose expected outcome is easily determined.

With every test case you will want to check the above criteria and perhaps some specific additional conditions as well.

Set-up of the application

The user's perspective

Of course, the application grew more or less organically, rather than from a deliberate design. Nonetheless, we can formulate a number of requirements, based on the likely users. We can not trust formal descriptions to be present and therefore can not rely on a full automation of the task (see below). We can however assume that any programmer who takes the job of testing seriously, can formulate a set of test cases and appropriate test criteria - whether these are sufficient or even suitable, is another matter. A programmer will want to test a reasonably sized part of the whole program, because otherwise formulating the test cases becomes a horrific job. Formulating the test cases and implementing them in a program should be easy to do.

The table below describes these requirements and others in some detail:

<i>Requirement</i>	<i>Rationale</i>
Test a reasonably sized part, not just the whole program	Large programs require a large set of test cases. Errors in the program will be difficult to trace because so much code is involved.
Concentrate on the specifications, not on the details of the test program	The test program's details are tedious and lead to errors. Small, clear test specifications reduce the work and the chance for errors.
Conclusions must be clear-cut	The test should either fail (with an indication of what criterion was violated) or succeed. Inconclusive tests are confusing.
Creating the program text should be automated and foolproof	Syntax and other errors in the user-supplied code are acceptable, but the generated code should be error-free.
The specifications should be regarded as data	If the programmer has to <i>program</i> the test cases and other information, chances are that he/she makes mistakes. The specifications should be as "data-like" as possible.

Design considerations

A good static analyser is capable of identifying the potential paths of control through the source code and generating a report of which conditions should be met. It is, however, not possible to derive the *test criteria* from this analysis. Consider the following, almost trivial, fragment in C:

```
void Increase( int *value )
{
    (*value) -- ;
    return ;
}
```

Judging from the *name* of the function one would suspect that the decrement ought to be an increment of the variable. But as long as there is no more or less formal description of this function that can be examined by some computer program, we will have to rely on our judgement.

Therefore the input for the TestMake application consists of a number of *user-supplied* code fragments that pieced together to form a complete program. The glue for this is provided by a standardised library of fragments, so that the user only has to specify the code that implements a particular test case and subsequently tests the results.

TestMake does provide some trivial but important automatic checks. As each parameter that is visible from outside the source code under test, is characterised as *input* or *output* in various flavours, the criteria below can be formulated:

- *Input parameters:*
Their value must not be changed as a result of invoking the code under test. To test this, a copy is made of their value just before the tested code is run and this copy is compared to the value upon return.
- *Output parameters:*
Their value must be changed, unless an error condition has been detected by the tested code. Testing that the value has indeed changed can be done in a similar way as for the input parameters, but their initial value must be set to something that is unlikely to be the result of exercising the tested code.
- *Input/output parameters:*
Some parameters will be updated by the code under test. Thus, they are essentially like output parameters and are treated this way by the automatic testing procedure.
- *Error parameters:*
It is assumed that an error condition has occurred whenever this type of parameter is set to a different value by the code under test. A correct detection of an error may be part of the test case, so this fact is simply reported and the automatic tests are influenced as indicated above.

In addition to these automatic criteria, the user will want to apply more specific tests. This can be done in two ways:

- As part of the definition of an output parameter
- As part of the definition of a test case

Skeleton code

To generate a full working program, a simple but effective approach is taken in TestMake:

- The user supplies the specific pieces of code, such as the declaration of a variable to be watched and the code for the test cases that are to be run.
- A library of small code fragments and auxiliary routines is used as a “glue” for all those pieces.

This way the test program always has the same structure:

```

Main program:
  (Program header)
  Headers
  Declarations of input, output and other parameters
  Additional declarations

  Preparation

  Repeat for all test cases:
    Call initialisation
    Set up the conditions for the test case
    Call the module
    Call the check routine
    Report the conclusions

  End of main program

Subroutine to initialise:
  Initialise all parameters
  Initialisation fragment

Subroutine to call the module:
  Call fragment

Subroutine to check the output:
  Repeat for each parameter:
    Generic check code
    Specific check code (if given)
  Report conclusion: test failed or not

```

By applying a different library (and perhaps redefining a few of the steps) one can generate programs in various languages.

In TestMake a whole set of small fragments is distinguished, so that it will not be necessary to redefine the above sequence for each and every language, though the implementation does assume that the programming language (or scripting language) allows for subroutines or methods and for multiple scopes of variables.

Let us examine one item in the construction of the final program more closely, the specification of an output variable. The user will have to supply the name and the type, an initial value (so as to detect the change as a result of the test case), and possibly a check on the resulting value. To take up the example again, the standard deviation might be defined as:

```

Output "vstd" "real" {
  vstd = -1.0 ! Should always be missing value, zero or positive
} {
  call test_failed( .not. (vstd .ge. 0.0 .or. vstd .eq. amiss),
&
  "Standard deviation negative" )
}

```

In the generated program, this information is used to:

- Declare the variable and a *copy* of that variable:

```

real :: vstd
real :: out__vstd

```

- Initialise the variable (and its copy) just before the code under test:

```

vstd = -1.0          ! In subroutine INITIALISE

```

```
out__vstd = vstd      ! In subroutine RUN_MODULE
```

- Check for the changes in the value as a result of the code under test:

```
if ( test_equals( vstd, out__vstd ) ) then
  write( test__lun, * ) "Output parameter vstd has NOT changed!"
  test__output = .false.
endif
```

- Check the value with the user-supplied code fragment:

```
call test_failed( .not. (vstd .ge. 0.0 .or. vstd .eq. amiss), &
  "Standard deviation negative" )
```

In a similar way, all other types of variables to be watched are treated and all code that prepares for the test cases is assembled.

Implementation in Tcl

As stressed in the requirements, the input, as far as the user is concerned, should look like *data* as much as possible. This can be achieved by defining the appropriate Tcl procedures. A bonus of this approach is that the user does not have to know Tcl and the implementation does not have to parse the data - it is simply *sourced*:

- All specifications are contained in an argument to the procedure *Module*:³

```
proc Module { module_name definitions } {
  global fragment
  global module
  global module_data

  set module      $module_name
  set module_data data $module_name
  upvar #0 $module_data a_name
  set a_name(name) $module_name
  set a_name(all)  {}

  set fragment($module,no_testcases) 0

  eval $definitions

  return
}
```

The *Module* procedure acts as a container for all specifications. That way it is always clear to what portion (module) of the code a particular fragment belongs.

- Variables are defined using procedures such as *Output*:

```
proc Output { varname vartype initcode { checkcode {} } } {
  global module_data
  upvar #0 $module_data params

  if { [ lsearch $params(all) $varname ] <= -1 } {
    lappend params(all) $varname
  }
  set params($varname,type)      "output"
  set params($varname,vartype)   $vartype
  set params($varname,initcode)  $initcode
  set params($varname,checkcode) $checkcode
  return
}
```

The element *all* stores the names of all variables defined in this way.

³ The code that is presented here does need some cleaning up.

- For automatically generating test cases from the information obtained by analysis tools, the *Condition* procedure has been defined:

```
proc Condition { expression value exprcode { checkcode {} } } {
    global module_data
    upvar #0 $module_data params

    set params($expression,$value)          $exprcode
    set params($expression,$value,checkcode) $checkcode
    return
}
```

There should be a specification for both the true and the false value of the condition.

- The user-supplied test cases are specified by means of:

```
proc Testcase { title code { checkcode {} } } {
    global module
    global fragment

    set no_testcases $fragment($module,no_testcases)
    set fragment($module,test,$no_testcases) $code
    set fragment($module,title,$no_testcases) $title
    set fragment($module,check,$no_testcases) $checkcode
    incr fragment($module,no_testcases)
    return
}
```

In contrast to the automatic test cases which can be constructed as the file with the information is examined, these “manual” test cases require extra administration, so that they can be recalled later on.

All the basic fragments in the library that provides the glue, are defined using:

```
proc Fragment { codename code } {
    global fragment
    global module

    set fragment($module,$codename) $code
    return
}
```

Because some degrees of freedom are required, for instance to get the names of the variables in, a number of reserved Tcl variables exist. For instance the fragment that declares a variable in FORTRAN 90 reads:

```
Fragment "declaration" {
    $vartype :: $varname
}
```

The variables *vartype* and *varname* are set and expanded when the fragment is written to the source file:

```
proc WriteFragment { codename } {
    global fragment
    global outfile
    global module

    set fragm \
        "[list subst -nocommands $fragment($module,$codename)]"
    set output [uplevel $fragm]
    puts $outfile $output
    return
}
```

The option *-nocommands* is necessary to avoid conflicting syntax in e.g. Java and C.

The example again

Given the above explanation of how *TestMake* has been set up, let us turn to the Java example again and see how this works out:

- The *preparation* fragment defines a new object that will be used throughout the test suite. The initialisation is meant to reset the status before each test.
- The *initialisation* fragment puts the object in its original state again, something which will be done before each test.
- The *input* fragment for the variable *values* prepares an array of values from which subsets will be used in the actual test cases. This is one way to deal with the need for various test data sets.
- Because the object will accept values one at a time and return the results only via *accessor* functions, the *call* fragment is more involved than its FORTRAN equivalent:

```
Call {
  ! Add the relevant data to the object and get the results
  call vstat( values(first:last), vmean, vmin, vmax, vstd )
}
```

- By setting the first and last indices into the array, each test case defines its own set of test data.
- The example does not include any *condition* fragments, the means to construct test cases from test paths. Currently, *TestMake* is somewhat limited in what it can do there (see the concluding remarks).

The module is specified by the following Tcl code:

```
Module "vstat" {
  Declarations { vstat v ; int first ; int last ; }
  Preparation { v = new vstat() ; }
  Initialisation { v.restart() ; }

  Input "values" {float[10]} {
    for ( int i = 0 ; i < 10 ; i ++ ) {
      values[i] = (float) i ;
    }
    values[0] = 1.0f ;
  }
  Output "vmean" "float" {
    vmean = -1.0f ;
  }
  Output "vmin" "float" {
    vmin = -1.0f ;
  }
  Output "vmax" "float" {
    vmax = -1.0f ;
  }
  Output "vstd" "float" {
    vstd = -1.0f ;
  } {
    /* If the standard deviation is not positive, failure! */
    Test.failed( vstd != missing_value && vstd < 0.0f,
      "Standard deviation is negative" ) ;
  }

  Call {
    /* Add the relevant data to the object */
    for ( int i = first ; i <= last ; i ++ ) {
      v.add( values[i] ) ;
    }
    vmean = v.average() ;
    vmin = v.min() ;
    vmax = v.max() ;
    vstd = v.stdev() ;
  }
}
```

```

Testcase "No data" {
    first = 1; last = 0;
}
Testcase "One single value" {
    first = 0; last = 0;
}
Testcase "Two identical values" {
    first = 0; last = 1;
} {
    /* Standard deviation should be zero, otherwise failure! */
    Test.failed( vstd != 0.0f, "Standard deviation should NOT have been zero" ) ;
}
Testcase "Two different values" {
    first = 1; last = 2;
} {
    /* Standard deviation should NOT be zero, otherwise failure! */
    Test.failed( vstd == 0.0f, "Standard deviation should have been zero" ) ;
}
# Etc.
}

```

General framework

The Tcl application that is presented in this paper can be regarded as an elaborate example of a whole class of applications. The characteristics of this class are:

- The need or desire to get rid of repetitious and therefore boring and error-prone coding tasks.
- The generation of programs or scripts in some language where the structure of the program is fixed.
- The program to be generated is composed of fixed fragments with only a few degrees of freedom.

Classic examples include, of course, Yacc and Lex, as generators of C programs that fulfil a certain limited task (still others are illustrated by Kernighan and Pike, ref. 3), but one can also think of:

- Simple forms of generic programming, when the programming language itself does not support that directly or only in a rather awkward way.
- XML data handlers - the if-constructs for branching to the proper tag would then be hidden.

Right now, TestMake has not been set up with such a more general approach in mind, but its elements could be used to create such a framework.

Concluding remarks

TestMake takes advantage of several characteristics of Tcl that make both its implementation and its use easier. With Tcl one can efficiently and effectively manipulate arbitrarily long strings. The input data can be shaped as Tcl procedures, so parsing the input becomes almost trivial. All code fragments are stored and handled via associative arrays. Whereas Tcl is certainly not unique in these respects, it proves a very useful tool.

In its present state, TestMake allows for a number of improvements. The script should perhaps generate a makefile for the test program, to facilitate this aspect as well, but right now, it is too restrictive with the construction of test cases from test paths identified by static analysis tools.

Work is now being done to create the more general framework that was mentioned in the previous section.

Literature

1. McCabe & Associates
User guide to the McCabe Visual ToolSets
2. P. Koopman and J. DeVale
The Exception Handling Effectiveness of POSIX Operating Systems
IEEE Software Engineering, volume 26, number 9, september 2000, pp. 837872
3. B. Kernighan and R. Pike
The Practice of Programming
Addison-Wesley, 1999



Creating generalised Tools for Database Access using Tcl/Tk

Matthias Lüttgert, RISA GmbH (<mailto:matthias.luettgert@risa.de>)

Dr. Johannes Heinrich Vogeler, Umweltbundesamt (<mailto:johannes-heinrich.vogeler@uba.de>)

Creating generalised Tools for Database Access using Tcl/Tk

Matthias Lüttgert, RISA GmbH, Berlin
Dr. Johannes-Heinrich Vogeler, Umweltbundesamt, Berlin

Up to two years ago we believed embedded SQL to be the most appropriate method for programming portable database applications although there are notable differences in the SQL description area (sqlda) for each database system and programming graphical user interfaces using OSF/Motif is very tedious. Besides that we had to consider that our applications should not only work with different database management systems but on different platforms as well.

In particular to provide versions for UNIX-Systems and MS-Window requires to maintain two different designed programmes for the same purpose ...

During the last years some powerful Tcl interfaces for the most commonly used database systems have been published; historically three 'interface families' have been evolved:

Family of oratcl interfaces

- | | |
|-------------------------|-----------------------------|
| • ADABAS-D | adabastcl |
| • ORACLE-7 and ORACLE-8 | oratcl (different versions) |
| • SYBASE | sybtcl |

Family of isqltcl interfaces

- | | |
|------------|---------|
| • INFORMIX | isqltcl |
| • MySQL | sql-tcl |

The odbctcl interface	odbctcl
-----------------------	---------

which from the point of view of a programmer are permitting a very similar access to databases running on different database management systems even for complex SQL statements as commands prepared at run time. Powerful Tk widgets allows for an efficient programming of graphical user interfaces for UNIX and Windows

Besides of that we found that data types implemented in Tcl are more suitable to handle records and intermediate tables. As an example let me illustrate the advantage of the list data type:

ESQL/C

- SQL is appropriate to describe sets
- C is used to handle single items of database records using host variables to be declared carefully

Tcl-Interface for databases

- SQL is appropriate to describe sets
- Tcl handles individual records and single items of those records using lists which need no prior declaration!

Furthermore there are other things as arrays of lists to handle result tables in a comfortable way; scrollable listboxes for representation of results ...

Conclusion: the Tcl interfaces for the database management systems are notable reducing the differences in database access for different database management systems. The power and versatility of Tcl encouraged us to write an abstraction layer for these database commands in Tcl – the DAC (= database abstract commands); we decided to restrict ourselves to classical database operations (select, insert, update, delete) for which the procedural/cursor oriented approach seems to be the most suited.

At the beginning there were abstract commands for INFORMIX, ADABAS–D and ODBC (having in mind MS–ACCESS and ORACLE). Revising our code we found that we had to think of a new architecture of our programme to take into account

- extensions for other database management systems
- modularisation for independent maintaince of different contributions
- unified error handling within an application – independent of the database system actual in use.

Namespaces turned out to be a adequate mean to make up a structure of our software, which corresponds to the requirements mentioned above.

`::dac::`

Toplevel, which also covers global data for every database session and some general procedures

`::dac::init`

load the tcl interface needed, set globals

`::dac::errmsg`

error handler (uses symbolic error codes
e.g. SQLE_NOTFOUND)

`::dac::cmds::`

database abstract command

`::dac::cmds::connect`

open a database session

`::dac::cmds::disconnect`

close a given database session

`::dac::cmds::open`

open a cursor for a database session and a statement

`::dac::cmds::close`

close an open cursor

`::dac::cmds::fetch`

fetch a record from an open cursor

`::dac::cmds::insert`

insert rows in a table

`::dac::cmds::update`

update rows of a table

`::dac::cmds::delete`

delete rows from a table

`::dac::cmds::commit`

make changes permanent

`::dac::cmds::rollback`

undo any change back to to last commit/rollback

`::dac::<database system>::`

database commands specific to a particular database system (ada := ADABAS–D)

`::dac::ada::fetch`

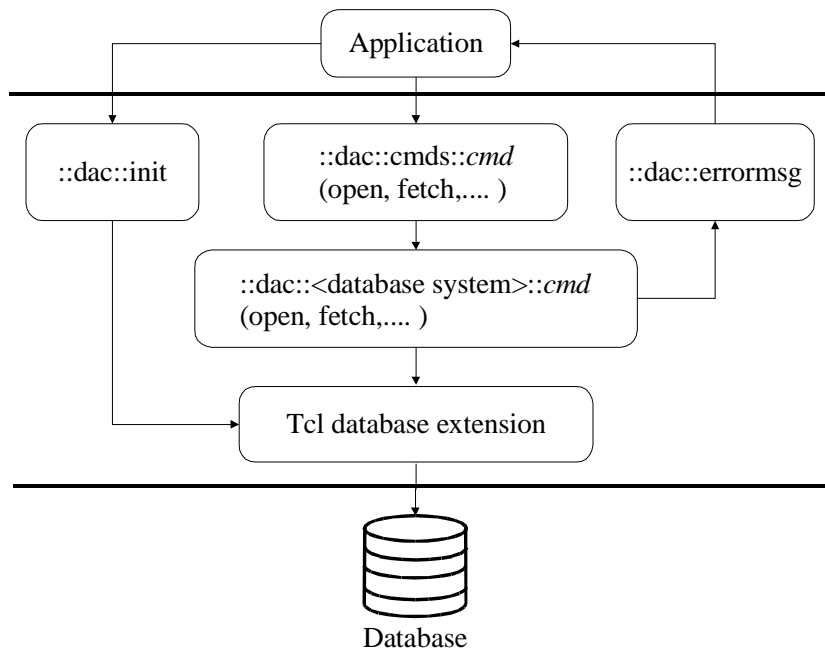
fetch a open cursor within an ADABAS–D session

these specific procedures may use also syntax checking and transform procedures we are keeping in the following utility namespace

`::dac::syntax::`

procedures for syntax checking and transformations (mostly quoting), normally called by the database specific procedures

At a first glance it seems a superfluous expense writing special procedures for inserting or modifying data in a table since the tcl-Interface of a database system allows to issue SQL commands passed as a string. In principle that is correct, however we found notable differences with respect to the syntax of the SQL commands passed as arguments and the handling of errors – and all that depending of the database system. We decided to hide these differences in the functions specific for a particular database system.



Although it has been our basic idea to get rid of problems arising from different structures of the sqlda, we found that sometimes there is a need to execute prepared commands on a native level. We therefore added the procedures

<code>::dac::cmds::prepare</code>	prepare a command at runtime
<code>::dac::cmds::exec_native</code>	execute a prepared command

Adding a library space

`::dac::lib::`
procedures for complex operations as e.g. 1toN relations

for database procedures, which are useful for solving common problems, this should be sufficient to write a portable database application using only Tcl/Tk

So far we have finalised the DAC for ADABAS-D, INFORMIX, ORACLE-8 and ODBC and written a complete application for a database of hydrological data in DAC.

But this is not all we are aiming for: a complete development environment requires further admi-

nistrative functionality to be added:

<code>::dac::cmds::get_tables</code>	get the table names belonging to a database
<code>::dac::cmds::get_table_info</code>	read the definition data of a table
<code>::dac::cmds::create_table</code>	create a table
<code>::dac::cmds::drop_table</code>	delete a table
<code>::dac::cmds::create_index</code>	create an index
<code>::dac::cmds::drop_index</code>	delete an index

Again at a first glance it seems that there is no need for these functionality since it is possible to issue a SQL script as a string to the Tcl–interface. Of course this works; however it is not tclish! In tcl it is far more convenient to use arrays and/or lists to handle these informations and to perform the operations mentioned above accordingly. We propose to proceed this way to build up a complete Tcl/Tk API for databases.

It is our intention to make this code available for the Tcl/Tk community in order to start an open source project for an easier way to write portable database applications and to demonstrate that Tcl/Tk is a language more suitable for that purpose than other classical languages as for instance C/C++. We already got the official permission to release this code under Tcl–License.

Roadmap



Using TCL as Middleware for Parallelizing Environment Development

M. Giordano (<mailto:M.Giordano@cib.na.cnr.it>) and M. Mango Furnari (<mailto:M.MangoFurnari@cib.na.cnr.it>)
Istituto di Cibernetica C.N.R.

Using TCL as Middleware for Parallelizing Environment Development

M. Giordano and M. Mango Furnari

Istituto di Cibernetica C.N.R.

Via Toiano, 6 I-80072 Arco Felice, Naples, ITALY

Phone: +39-81-8534229/227

{M.Giordano,M.MangoFurnari}@cib.na.cnr.it

Abstract

Software Development Environments (SDE) are typically software systems equipped with a collection of integrated tools to assist the programmer in the software development and/or maintenance. SDE design in parallel programming is much more complex since the parallelization of large application requires a deep knowledge of the code structure and data-usage to individuate parallelism sources as well as sophisticated compiler technologies and runtime support to efficiently exploit the discovered parallelism on the target parallel architecture. Therefore, the parallelization of large applications requires the support of interactive compilation environments, equipped with program structure visualization and interactive tools, to better exploit both user and compiler knowledge to drive the parallelization process. In this paper we describe the *Graphic Parallelizing Environment* (GPE), a unified environment for parallel program development based on user interaction with different components: compiler modules providing specialized services (optimizations, analyses, etc.), visualizers of program representations, smart editors and modules for the parallelization control. The GPE uses TCL as middleware for the integration of its components as well as for component interoperability and data exchange. Moreover TCL/TK serves as development environment for the GPE graphic tools that implement user interaction in terms of program structure visualization, parallelism control and syntax oriented editors.

1 Introduction

The term *environment* is typically used to describe an integrated collection of tools that assist the programmer in developing and/or maintaining software [1]. Though historically derived from a collection of independent programs, compilers, debuggers, etc., the *Software Development Environment* (SDE) attempts to be more than the sum of its parts. Its goal is to simplify and speed the development process through a tighter integration of the underlying tools, taking benefits from user interfaces, control integration and data sharing among the tools.

This scenario is much more complex for the development of parallel applications. The needs of an efficient SDE in parallel programming is strongly pushed on the fact that parallel processing, which was originally conceived for high-end systems, is currently migrating to low-cost workstations. This trend will continue towards PC-systems in the future. Programmers for these architectures find significant difficulties in getting the best performance out of their programs. Base software (including compiler and runtime support) for these architectures lacks of the required functionality to exploit all the parallelism available in the application in particular when running in a multiprogramming setting.

Therefore, the parallelization of large applications requires the support of interactive compilation environments, equipped with program structure visualization and interactive tools, to better exploit both user and compiler knowledge to drive the parallelization process. Graphical representations of program structure, control flow, and data usage have always been part of the programmer's repertoire of tools and techniques. Such representations can simplify and enhance the explanation of specific aspect of a program and aid the user in understanding the parallelism it is worth to be exploited at program execution.

In this paper we describe the *Graphic Parallelizing Environment* (GPE), a unified environment for parallel program development based on the interaction of user choices and compiler techniques for an efficient program parallelism exploitation. A program parallelizing environment based on this approach requires the integration of different components: compiler modules providing specialized services (optimizations, analyses, etc.), visualizers of program representations, smart editors and modules for the parallelization control.

We choose TCL/Tk as integration language for those components. The TCL interface towards the C programming language allows to easily design wrappers to translate internal data structures of existing environment modules into TCL data. In this way the TCL middleware represents the *shared arena* for operating on and exchanging data among different interacting modules.

The paper is so organized: section 2 gives an overview of the GPE design architecture with a brief description of GPE main components; section 3 show more details about component interaction, and data structures used for communication during a GPE session; section 4 is a overview on some related works, while section 5 reports some concluding remarks.

2 The GPE architecture

The basic idea of the GPE is that program parallelization is the result of a cyclic process in which, at each round, the user has a primary role interacting with the compiler and the other tools of the GPE. User actions includes to refine task partitioning, decide task and loop parallelization according to compiler analysis results and previous user specifications, drive parallel code production for program units and mark code to be instrumented for execution.

During a GPE session user-compiler interaction is implemented around the notion of program *task graph* (TG). We define the program TG as a directed graph whose nodes are *tasks*, that is program computational units like *statements*, *loops*, *subroutine calls*, or *basic blocks*. The arcs in the graph impose sequentiality among tasks, and the execution order of tasks result from data dependences analysis. Then, the *task parallelism* is expressed by the graph structure: branching points individuate program units that can be executed in parallel.

During compilation in a GPE session, the user may interact with different views of the program TG and with other textual representations: he (she) gets information like concurrency and *data* and *control dependences* [5] among tasks detected by the compiler; the user may also interact with these graphic views to impose task granularity, task parallelism (or serialization) and remove unwanted dependences.

User intervention is carried out by means of OPENMP directives [13] injection into the intermediate code, that is textual changes that the compiler uses to recompute dependence analysis and to build the task graph according to the user decisions.

after compilation and program execution the user may gather running profiles of the parallel code and use this information in the next round of the tuning cycle, in a feed-back process that should converge to the optimal solution.

In what follows we summarize user activities during the GPE session together with the tools used to perform them:

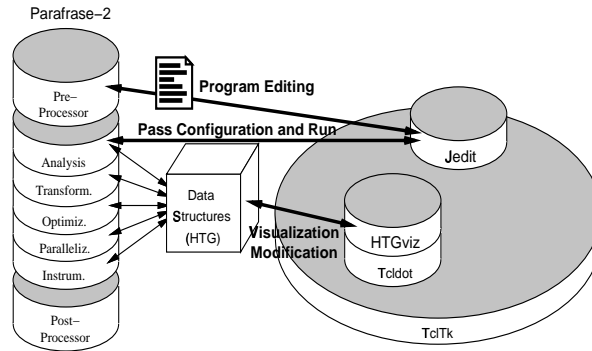


Figure 1: GPE architecture

- program editing and first parallelism specification by means of directive insertion (JEDIT);
- compilation process configuration and driving (JEDIT + PARAFRASE-2);
- visualization of program task graph parallelism (either specified by the user or detected by the compiler), task dependences visualization (HTGVIZ);
- program task graph parallelism tuning, by selecting tasks and, then, inserting OPENMP directives in the task-associated code (HTGVIZ);
- selection of parallel code production and instrumentation, by marking tasks and, then, activating code generation passes (HTGVIZ + PARAFRASE-2);
- program execution (JEDIT).

Figure 1 describes the GPE architecture, its components and how they interact. GPE components are described in more details in the following subsections.

2.1 The parallelizing compiler: PARAFRASE-2

PARAFRASE-2 [15, 16] is a source-to-source multilanguage restructuring compiler. It provides a reliable portable and efficient research tool for experimenting with program transformations and other compiler techniques for parallel shared memory supercomputers. Figure 1 shows the different components of the PARAFRASE-2 framework. PARAFRASE-2 uses an aggressive approach for dependence testing [4], including traditional tests as well as symbolic dependence analysis techniques.

The compiling process is carried out by applying several intermediate *passes*. Each pass implements a specific compiler activity, like the production of program *control flow graph*, the *induction variables transformation*, the *code generation*, the *code instrumentation*, and so on. Each pass operates on the internal data structures to transform the program to a form suitable for the next pass. Passes can be executed in any sensible order; this is achieved requiring that the form of the data structures is left invariant by each pass. The output of PARAFRASE-2 is the modified version of the input program representation, which is acted on by a post-processor to produce the required output language of the original one. The PARAFRASE-2 passes are grouped into the categories shown in Figure 1.

As already mentioned, in the GPE the compiler-user interaction is implemented around the notion of program *task graph* (TG): we are convinced, indeed, that the program TG could be considered the program representation closer to the user conceptual program representation in the case of parallel execution. The program TG is also used by PARAFRASE-2 as intermediate program representation to express and synthesize all parallelism discovered by the compiler analyses passes. The PARAFRASE-2 approach to task formation is based essentially on the syntax of the underlying language. We define

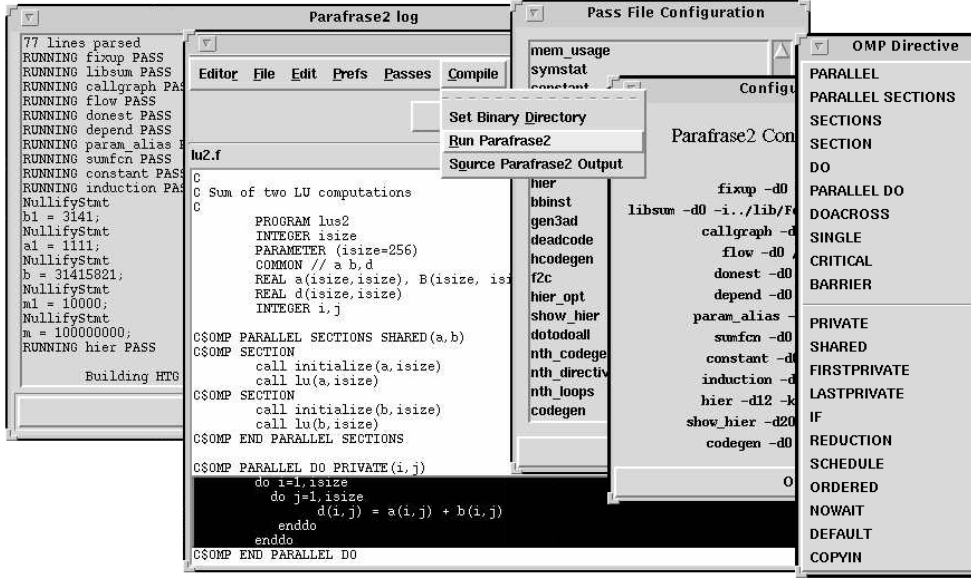


Figure 2: The JEDIT interface

as *task* a section of code delimited by “natural” boundaries, such as a *statement*, *loop*, *subroutine call*, or a *basic block*. Nodes in the task graph correspond to tasks, and an arc implies the existence of one or more *precedences* between two tasks. During execution, a task cannot start unless all the preceding tasks, which it depends on, have completed execution.

If we represent each loop in the TG as a unit of potential parallelism, and we consider that the loop body can be composed of other sub-units, like basic blocks and other loops, in an arbitrary number of nestings, we understand that the structural complexity of the task graph could overcome the users capability. To manage this situation PARAFRASE-2 adopts as program representation a hierarchical version of the task graph, named *Hierarchical Task Graph* (HTG) [5, 6, 7, 8]. It must be observed that the HTG implicitly contains all parallelism in a given program, from coarse- to fine-grain parallelism.

In the GPE we used an extended version of the PARAFRASE-2 compiler, (the NANOSCOMPILER¹) where the FORTRAN front-end has been modified to parse OPENMP directives. Also the compiler passes, dealing with task formation and HTG building, were changed to account for user directives. Directives express how the user has partitioned the program into tasks, at the coarser granularity, and how they have to be executed (concurrently or serially) and synchronized. The compiler builds the HTG and the other representations (like *data* and *control dependence graphs* [5]) in such a way they faithfully implement user choices.

With this approach, the compiler assumes that all user choices are “correct”. Therefore, the user is responsible for the correct program execution through a careful use of directives. Once the user has fixed the HTG lower hierarchical level (coarser tasks), the compiler can apply transformation, optimization and analysis passes to parallelize the code, and, then, refine the HTG inside the user-defined task. Therefore, the HTG synthesizes both user parallelization, expressed by input OPENMP directives, and automatic detected parallelism, coming from compiler dependence analysis.

2.2 The extensible editor: JEDIT

The JEDIT text editor application is a customizable multi-mode X Windows text editor. It is distributed as part of the JSTOOLS package, a suite of applications written in TCL/Tk and some libraries that they share. The JSTOOLS distribution (applications, libraries, and support files) is copyrighted by Jay Sekora [19].

JEDIT is intended to be flexible and configurable. Preferences panels let users to choose among sets of text bindings, including EMACS and rudimentary VI emulation, and general aspects of the application behavior. Furthermore, hooks are provided to let a further customization application behavior via startup files written in TCL. The editor supports a number of keyboard shortcuts for menu commands, as well as a few keyboard shortcuts for buttons in dialogue boxes. The editor can be incorporated into other TK-based applications as a set of libraries.

JEDIT supports the notion of distinct editing modes for different types of file. There are a few modes distributed with the editor, and you can implement additional modes, or modify the existing ones. The editor will behave differently depending on which mode it is in. For instance, an editing mode for C code might try to automatically change the indentation of each line based on the structure of the code, or a mode for TEX source might display TEX keywords in a different font from ordinary text.

We extended JEDIT by creating a new editing mode that inherits all the functionalities of the default mode and adds new functionalities oriented to the JEDIT use within the GPE environment. We implemented a new functionality to easily manage OPENMP directives insertion into the edited file (see Figure 2). From this menu it is possible to select the directive (or clause) we want to insert on the text selection. A syntax control minimize the number of writing errors that are often very frequent in the “hand” typing of programs and are responsible of most compilation errors.

The main functionalities are those implementing the compiler passes configuration and control. Figure 2 shows an example of compiling the LU2 program with the configured passes listed in the rightmost window. The JEDIT interface provides an easy way to add and remove passes from the configured set and specify pass arguments and options. Once a pass has been selected, the list of all passes it depends on is automatically included in the set of configured passes. This feature helps the user in having a clear idea about which transformations (optimizations) are applied and in which order.

Once the pass configuration is chosen, the user can go from the editing phase to the compilation. Compiler passes execution is controlled from the JEDIT interface. After the compilation is finished, the user has two choices: he (she) may compile again, specifying new (or the same) passes, otherwise he (she) may load the result of the source-to-source compilation, edit it, compile it, and so on.

2.3 The program task graph visualizer: HTGVIZ

The program task graph visualizer (HTGVIZ) is automatically activated when the pass `show_hier` is selected for compilation. HTGVIZ has been designed to offer different HTG views concerning:

- *program partitioning* into tasks of any granularity, visualizing the HTG structure across hierarchy levels and the code associated to HTG nodes. This view is useful to discover the best program partitioning into tasks according to the user-chosen granularity.
- *intertask dependence* analysis, visualizing, in the HTG, task sequential execution order, with *control* and *data dependences* among tasks. This view is useful to discover, while looking at the corresponding code, unnecessary or removable dependences that inhibit parallelization, information otherwise hidden to the user.

¹developed within the Esprit Project NO.21907 - <http://www.ac.upc.es/nanos>

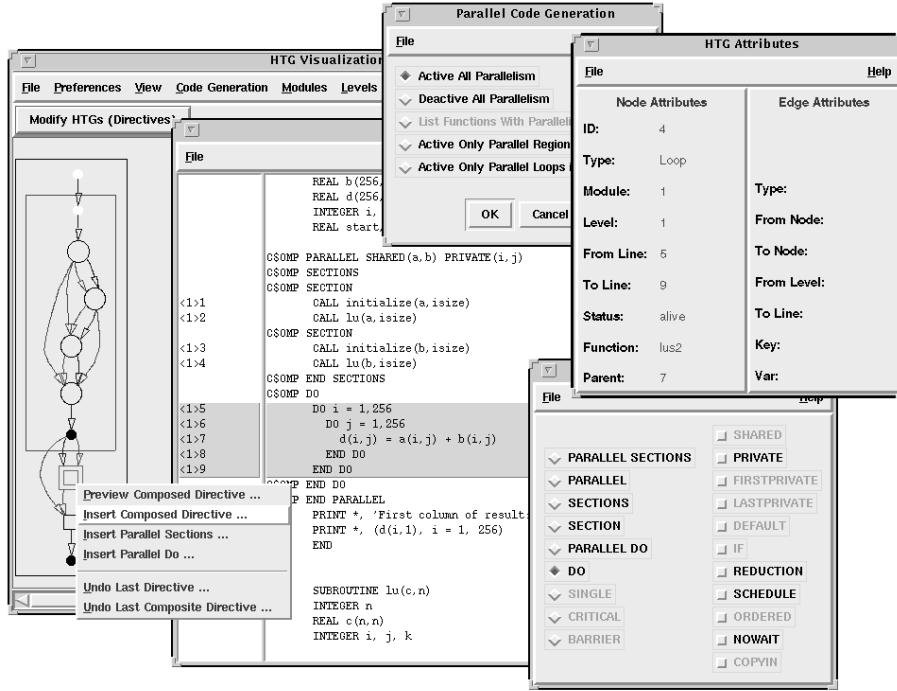


Figure 3: HTGViz interface

- *program parallelization*, visualizing, in the HTG, task parallel execution order showing only precedence relations among tasks. This view gives a complete overlook of all program tasks, discovered by either the compiler or the user (directives), that could be executed in parallel, at each level of the hierarchy. This will enable the user to exploit only some parallelism and decide the sequential execution of other tasks.

These different views offered by the HTGVIZ may serve different kinds of users: some of them may be more interested in the program task partitioning or dependence analysis performed by the compiler; other users may want to focus on the program parallelization resulted either from the use of directives or from the compiler analysis. The HTGVIZ is composed of five main user interfaces showed in Figure 3:

- **Task graph visualization window** - Here the user navigates the HTG through hierarchy levels. For each program module (main, subroutine, or function), the user may expand (collapse) the HTG on single nodes or as a whole: in this way he (she) may set program partitioning grain towards parallelization. Several events are allowed on the HTG view: selecting nodes and arcs to gather information about the related code, dependence information, and so on; selecting nodes for OPENMP directive insertion in the text of the program intermediate version; alternatively, hide/show control flow arcs, precedence arcs, data and control dependence arcs to switch from one HTG view to the other.
- **Program code window** - Program code lines are uniquely indexed by means of the belonging module identification number, and a line counter related to the first line of that module. This interface was designed to always show the correspondence between program statements and HTG nodes during all user actions, like HTG navigation, browsing, and directive insertion.
- **Task graph attributes window** - This interface reports all useful HTG information, like node type (loop, basic block, and so on) and arcs type (precedence,

data dependence, and so on). Another important information for dependence arcs is the set of variables causing the dependence.

- **Directive composer window** - This interface was designed to help the user in composing directives to be inserted, by specifying clauses and arguments for those clauses. The composer has a syntax control that inhibits insertion of wrong formed directives. Once the HTG nodes (and their code) are selected and the directive composition is complete, the directive is ready to be inserted. After directive insertion, the user may force the compiler to produce the HTG changes caused by directive insertion.
- **Parallel code & instrumented code windows** - From these interfaces it is possible to drive the code generation step: the user selects the program units (parallel sections and loops), annotated with OPENMP directives or parallelized by the compiler, for which to produce parallel code and instrumentation. Instrumented code is used to get detailed information about performances of the program parallel execution.

3 The GPE middleware: TCL/TK

The GPE uses the TCL/TK as middleware for the integration of its components as well as for component interoperability and data exchange. In fact, interoperability is easier within the TCL scripting environment since here it is easier to manipulate the large amount of symbolic information that is produced and used during the compilation process. Some components require the same information to be computed in order to provide their services: the use of a *shared arena* allows to produce only once this information and make it permanent and accessible to all the components that request it. In the integration layer, information coming from different components can be easily compared and combined to produce newer and more sophisticated services.

The GPE core component is PARAFRASE-2 whose architecture is modular, that is the compiler passes are implemented in modules (subroutines). Since it is easy to extend the TCL language adding new commands associated with C or C++ procedures, the result was the compiler split into a set of procedures (passes) imported by the TCL interpreter as new commands. When the GPE starts a TCL/TK interpreter is started and it runs and interacts with the compiler modules as long as the GPE session is alive.

Graphic components, like the task graph visualizer (HTGVIZ) and the editor (JEDIT) for program editing, compiler passes management and OPENMP directives insertion, are implemented in the TCL/TK environment. While HTGVIZ is a new developed component of the GPE, JEDIT is a software package developed by other research groups, integrated into the GPE and extended with new editing modes and functions. The development of GPE proved us that TCL is a good platform for fast prototyping of new software components as well as for integration and re-use of existing software.

In the GPE all the compiler environment (data structures and procedures) is visible to the TCL/TK interpreter. Therefore all graphic components can access compiler data structures and modify them. Therefore, the original JEDIT editor was extended to provide the user with an interface for the configuration and execution of compiler passes. The user may interactively decide which is the next (set of) pass(es) to be (re)applied. JEDIT interacts with the compiler front-end module by means of configuration files. These files register the compilation pass list as well as all the specified options for each pass. Some options are for instance, debugging level, trace files, program analysis tuning options, program transformation and optimization switches, and so on.

The second JEDIT extension was to provide the user with smart editing facilities towards OPENMP directive insertion and automatic check of directive syntax. This functionality was implemented using the regular expression processing capability of TCL. The use of regular expressions in pattern matching and substitution is a powerful and efficient way for string processing which is the basis of syntax oriented editors.

Most of compiler-user interaction takes place around the notion of program HTG. HTG visualization serves to help the user in understanding program structure, control flow and data-usage to discover parallelism sources. The HTG is also a mean for the user to impose program parallelization via graph manipulation and directive injection into the original code. Therefore the main data structure to be visualized in the GPE is the program HTG.

The HTGVIZ tool was designed mainly as an user interface for program parallelization tuning and as a visualizer for the program HTG produced by PARAFRASE-2. The HTGVIZ is implemented in TCL/Tk using a library, named TCLDOT, that adds graph manipulation facilities to TCL/Tk. TCLDOT is part of the GRAPH Visualization Software (GRAPHVIZ) package² [12].

The HTG and all its information (internal to the compiler) are dumped into the TCL environment: here a directed graph in DOT format is created and enriched with annotations to reproduce all the information and the structure of the HTG built by the compiler. This graph is then showed by the HTGVIZ to the user in an eye-pleasant manner together with the corresponding program code in textual format. The user acts on the HTG dump and, by means of graph manipulation he (she) selects parts of code and add OPENMP directives to overwrite the parallelism specification. Once the changes are done the compiler processes the modified intermediate code once again building a new HTG which faithfully reproduces user actions, and so on.

4 Related works

Graphic program display have been developed in the parallelization context mainly to analyze and to assist the users on the interpretation of parallel program execution performances. In this case the purpose of a visualization tool is oriented to debug a program, to evaluate or optimize performance, to provide a form of data structure display, display that conveys knowledge of the functioning of a program. Such performance evaluation systems analyze the raw data, collected during the parallel program execution, to determine overall performance metrics and statistics. Systems, such as PARAGRAPH [11], analyze, for example, the message traffic coming up from the execution of parallel programs on a distributed memory machine, matching send and receive events, keeping track of message counts and volumes. These events are used to create higher-level visualizations of utilization, computation and task information. Task displays in PARAGRAPH are program graphs in which the nodes represent program entities and the arcs represent call relations or temporal ordering.

STAT/PAT [3] is an interactive toolkit for the development and debugging of multi-tasking FORTRAN programs. *Dependence graphs* are generated in which a node represents a variable access, and an arc connects two accesses whenever a write access can reach a read access. These nodes are labeled with source line number, the variable name and subscripts, and a notation of whether the access was read or write. A *concurrency history graph* represents the reachable states in the execution of the parallel program and can be used to detect data-usage and deadlock errors.

In [2] an integration of the PABLO *Performance Environment* [17] with the FORTRAN-D compiler environment is described. The integration relies on extending the PABLO environment to support abstract parallel languages, and, for the compiler, to export the necessary information to correlate dynamic performance data with the program source code.

PARASCOPE Editor [10] is an interactive parallel programming tool that assists users in developing scientific FORTRAN programs. It displays the results of sophisticated program analysis, provides a set of powerful interactive transformations and supports program editing.

²developed at the AT&T Laboratories and Bell Laboratories (Lucent Technologies)

SUIF EXPLORER [14] is another visualization tool, similar to PARASCOPE Editor for the goal pursued. It is an interactive parallelization tool based on interprocedural parallelization techniques and dynamic execution analyzers to identify the important loops that are likely to be parallelizable. Furthermore, SUIF EXPLORER uses the *program slicing* concept to identify the subset of statements that is relevant to determine if a data dependence exists in a loop. The results of these analysis are presented to the user through sophisticated displaying tools.

NARAVIEW [18] is a parallelizing environment where program visualization tools are used to illustrate the program structure and data dependences. The *program structure view* illustrates the hierarchical loop structure of a given program and suggests which parts of the program can be parallelized. The *data dependence view* visualizes each data dependence on every variable or array element which is access in a specific loop. By using these views, users understand which part of the program can be parallelized.

5 Conclusions

GPE is an interactive compiling environment equipped with graphic tools to exploit user knowledge and compiler techniques for the efficient parallelization of sequential programs. The core of the system is PARAFRASE-2, a source-to-source parallelizing compiler, while TCL/Tk is the integration language for components interaction, user-system interaction and graphic processing. The graphic tools of GPE are: JEDIT, a program editing tool with OPENMP directives insertion facilities, with an interface to control and manage compilation passes execution; HTGVIZ, a tool to visualize the parallel program representation (*task graph*) and data dependences, with an interface for parallelism tuning and task granularity control.

In the GPE interoperability is afforded by the TCL scripting environment: with TCL it is easier to manipulate the large amount of symbolic information that is produced and used during the compilation process. Some components require the same information to be computed in order to provide their services: the use of a *shared arena* allows to produce only once this information and make it permanent and accessible to all the components that request it. In the integration layer, information coming from different components can be easily compared and combined to produce newer and more sophisticated services.

In our experience TCL/Tk proved to be a good platform for the design and implementation of a software developing environment like the GPE. In fact, in the TCL/Tk environment it is easier to carry out activities like the fast development of new software graphic components as well as the integration and re-using of existing software packages.

When developing complex graphic user interfaces (GUI) like the HTGVIZ, the use of TCL/Tk gives the possibility of quick assembling graphic objects (widgets) and binding commands to them for the fast prototyping of the interfaces that make up the GUI. On the other hand, as the GUI code get larger, TCL/Tk code becomes more and more difficult to understand, maintain and extend. The use of INCRTCL in a future development of GPE will offer the object-oriented abstraction as a powerful support for developing large GUI applications.

When developing large application you can also take advantage of software re-use, as we did with the JEDIT software, especially when you may choose among lot of software products available in a large developer community like the TCL one. This is even more true when embedding third party TCL libraries and packages into your application is an easy task.

References

- [1] Adams R., Tichy W., and Weinert A. The Cost of Selective Recompilation and Environment Processing. ACM Trans. Soft. Eng. and Meth., 3, 3-28 (1994).

- [2] Adve V.S., Wang J.C, Mellor-Crummey J., Reed D.A., Anderson M., and Kennedy K. An integrated Compilation and Performance Analysis Environment for Data Parallel Programs. CRPC-TR94513-S (1994).
- [3] Appelbe B., Smith K., and McDowell C. Sart/Pat: A parallel programming toolkit. *IEEE Software*, 6 (1989), 29–38.
- [4] Banerjee U. Dependence Analysis for Supercomputers. Kluwer Academic Publisher (1988).
- [5] Girkar M. Functional Parallelism: Theoretical Foundations and Implementation. PhD Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign (Urbana IL, 1992).
- [6] Girkar M. and Polychronopoulos C.D. The HTG: An Intermediate Representation for Programs Based on Control and Data Dependencies. CSRD TR 1046, Univ. of Illinois at Urbana-Champaign (Urbana IL, 1990).
- [7] Girkar M. and Polychronopoulos C.D. Automatic Detection and Generation of Unstructured Parallelism in Ordinary Programs. *IEEE Trans. on Parallel & Distributed Processing* (1992).
- [8] Girkar M. and Polychronopoulos C.D. The Hierarchical Task Graph as a Universal Intermediate Representation. *Int. J. Parallel Programming*, 22 (1994), 519–551.
- [9] Hall, M.W., Anderson, J.M., Amarasinghe, P., Murphy, B.R., Liao, S.W., Bugnion E., and Lam, M.S. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer* (December 1996).
- [10] Hall M.W., Harvey T.J., Kennedy K., McIntosh N., McKinley K.S, Oldham J.D, Paleczny M.H., and Roth G. Experience Using the ParaScope Editor: an Interactive Parallel Programming Tool. In *Proceedings of the Symposium on Principles and Practice on Parallel Programming* (San Diego, CA, May 1993).
- [11] Heath M.T., and Etheridge J.A. Visualizing the performance of parallel programs. *IEEE Software*, 8 (1991), 29–39.
- [12] Krishnamurthy B. Practical Reusable Unix Software. John Wiley Sons (1995).
- [13] OpenMP Organization, Fortran Language Specification, v. 1.0. <http://www.openmp.org/openmp/mp-documents/>. (October 1997).
- [14] Liao, S.W., Diwan A., Bosch R.P., Ghuloum A., and Lam M.S. Suif Explorer: An Interactive and Interprocedural Parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, Georgia, May 1999).
- [15] Polychronopoulos C.D., Gyrkar M.B., Haghighat M.R., Lee C.L., Leung B.P., and Schouten D.A. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. *Int. J. of High Speed Computing*, 1, 1 (1989).
- [16] Polychronopoulos C.D., Gyrkar M.B., Haghighat M.R., Lee C.L., Leung B.P., and Schouten D.A. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran. *Languages and Compilers for Parallel Computing*, MIT Press (1990).
- [17] Reed D.A., Aydt R.A., Noe R.J., Roth P.C., Shields K.A., Schwartz B.W., and Tavera L.F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *IEEE Proceedings of the Scalable Parallel Libraries Conference*, Ed. Skjellum A. (1993), 104–113.
- [18] Sasakura M., Joe K., Kunieda Y., and Araki K., NaraView: an Interactive 3D Visualization System for Parallelization of Programs. *ISHPC' 97, Lecture Notes in Computer Science* 1336 (1997), 231–242.
- [19] The Jstools Application suite and libraries. <http://www1.shore.net/~js/jstools/>



Tcl on the iPaq

Lindsay Marshall (<mailto:Lindsay.Marshall@newcastle.ac.uk>)
Dept of Computing Science
University of Newcastle upon Tyne

Tcl on the iPaq

– A Report –

Lindsay Marshall

*Dept of Computing Science
University of Newcastle upon Tyne
UK NE1 7RU*

Abstract

The use of Linux on PDAs is currently a hot topic for discussion and research. A lot of application development that is being done in this area uses either Java or Python. In this paper I look at using tcl/tk as a development language and suggest that it may be more suitable than either of the more frequently used languages.

Introduction

There has been much talk recently of a new breed of PDAs running Linux instead of PalmOS or Windows CE. The Agenda is on example and Sharp are promising a Linux based machine as well. However, for some time now the Linux-based PDA has been a reality, albeit one that costs rather more than a Handspring or Palm Pilot for the basic hardware. There are several hardware bases that people used, but the most common is the Compaq iPaq – a handheld with a 240x320 16 colour display, upwards of 16M of RAM and 16M of flash, powered by a 206 MHz StrongARM CPU. The iPaq can be expanded using a sleeve system and so can have PCMCIA cards connected to it.

The iPaq comes with WIndows CE loaded on it, but for an experienced computer user it is a (fairly) easy matter to replace this with one of several Linux distributions. These distributions all use the same basic port of the 2.4 kernel to the ARM processor, but come with different sets of applications and file system layouts. I have been experimenting with the Familiar distribution which in its recent versions supports a journaling file system – particularly useful for flash memory which can only carry out a finite number of writes before it fails. The operating system has drivers for a wide variety of add-ons and wireless networking worked straight away for the card the Wavelan card I use.

The applications that come with Familiar vary from version to version but they usually include a shell (ash), the X Window System, the iv editor and Python. (The very latest version uses a package management system that allows control over what is installed.) Many of the Familiar team are Python aficionados and their thinking is very much geared towards using Python to produce a set of PDA applications for, as it stands, an iPaq running Familiar is not especially useful as a PDA.

Naturally, being a tcl devotee, I see things differently and it was instantly obvious to me that using tcl on the iPaq would be a great improvement over Python. For a start the basic interactive tcl interpreter is a much better shell than ash! I tested the waters by asking the mailing lists about tcl and uncovered three other people using tcl, and one person who was said that they had tried it but that it was painfully slow – this did not square with the experience of the other users so I pushed on.

The Basic Port

The first requirement was a version of *wish*. I could probably have copied someone else's binary but as I wanted the possibility of playing around with extensions I thought that I had better build it myself, which experience of tcl suggested would not be a problem. And it was not a problem – once I had managed to get hold of a working arm cross-compilation system and copied all the necessary libraries from the iPaq up to my development platform. Many iPaq developers use a system provided by Compaq for their compilations but my experience of trying to use this was not positive so I decided on a standalone approach.

With all the tools in place the standard installation worked smoothly and, when everything was copied across, *wish* worked straight away and has yet to fail for me. This speaks volumes both about the quality of the tcl/tk distribution and of the porting work on the iPaq, especially the X Windows port.

First Steps and Stumbles

After playing around with the system a little I decided that I needed to give it a thorough work out so I loaded my Zircon IRC client and ran that. And I had the first spot of trouble. Not, I must stress with tcl, but with the pen input system that comes with the X distribution. I simply could not get it to give me a '.' and so I couldn't setup the IRC server name I wanted to connect to! (At this stage I had dismally failed to work out how to use the on screen keyboard application as well so that was no help either.)

Using IRC on my development machine I managed to find the right people to ask and was pointed at an improved pen input program (xstroke) and told how to use the screen keyboard, so, better armed, I re-ran Zircon and made a connection. The speed of the wireless connection was not a problem, the pen input wasn't much of a problem, but the big windows that appeared certainly were. The standard Zircon channel window was set up for a much larger screen and needed to be drastically shrunk. The width of the windows is mainly controlled by a row of control buttons and by reducing the size of the default button text typeface it was possible to shrink the window considerably and maintain readability. However the default portrait orientation of the screen still prevented me from seeing the full width of the window. The Familiar distribution does not seem to support the ability to rotate the screen orientation at the moment (some other distributions do) and the kind of textual interaction that takes place on IRC really needs the wider screen that landscape would allow. This is an important issue that application developers need to take into account if they want their programs to be portable as possible across the range from PDAs to desktops.

I had been told by other tcl experimenters that some of the standard dialogue boxes needed to be resized and this was indeed the case. The two most difficult problems were the error reporting window, which I did eventually manage to make usable, and the file selection widget. This latter has so far resisted all my (fairly feeble) efforts to get it to fit nicely on the portrait screen – again landscape would make all the difference here. The 8.4 implementation is very different from that in 8.3 and both are pretty complex and quite hard to get grips with. My aim has been to try and preserve the look and feel of Unix tcl/tk and it may be that this is not the way to proceed. After all, on other platforms the native file dialogues are used, so inventing a new layout for small screen systems should not present too much of a problem so long as it maintains the functionality. Once again it is the layout of the various buttons that the dialogue offers that present the biggest problems.

The only other area where there are problems is that of default sizing. The built-in, default sizes for some widgets are too large for small screened systems and there seems to be no easy way to configure these defaults when building the interpreter. I'm sure that it is not particularly hard to change these, but if the standard distribution could collect all these values together in one place where they could be changed consistently, or perhaps have `#ifdefs` for various systems, life would be much easier. (Of course, there probably already is a central place with all this information, in which case it is the documentation that needs improvement...)

In general the typefaces on the iPaq under Linux are not great, but work has been done to port Truetype support and to add anti-aliased fonts into tcl. This apparently is a major improvement, but I have not yet seen a system running this way yet so cannot give an opinion.

Experiments

When you boot the iPaq, it starts up the X Window System for you automatically and runs the Blackbox window manager. This is fine, but is it necessary? Do you really need a window manager on such a small screen? I tried running *wish* directly on top of X and it works fine. It would be perfectly possible to build a suite of PDA applications that all ran inside a single instance of wish and by keeping all the individual application toplevel windows at full screen size there would be no need for a window manager at all. (The other problem with window managers is that they tend to divert people from the task in hand – making a useful PDA – towards an obsession with themes and other such frills)

The snag with doing this is the problem of character input. The standard iPaq input method is via the pen and users expect some kind of limited handwriting recognition – having a keyboard widget that pops up and down is, of course, not hard to do. Two solutions present themselves. The first is to reserve a chunk of the screen for character input as the Palm Pilot does which is relatively easy, but is a great waste of screen real estate. The second would be to allow characters to be written on the screen freely, which is what the scribble and xstroke applications do at the moment.

I implemented a simple stroke recogniser in pure tcl that works inside a canvas, and then tried to extend this to work freely on the screen. Unfortunately the low-level window clipping stops this from the working as

the mouse trace is not displayed which is not exactly user-friendly. After some discussion with others, I tried to work out a solution that used the tk interface to the X shape extension to allow me to have “invisible” windows. So far, however, I have had little success, though I did make some nice round windows. The other problem here is speed – it may be possible to make this solution work, but it would probably require a lot of the stroke recognition to be moved into C as an extension.

I don't know if the rootwm patch to tk would allow some other way of approaching the problem, but it has not been ported to version 8.3 so it would have been a backward step to spend too much time with it. (The patch kit is for 8.2 and it will not work with 8.3)

The Way Forward

Linux on handheld computers is most definitely a growth area. There is a considerable amount of consolidation amongst the various development teams which are trying to merge their efforts in various ways. At the moment, the Python and Java users seem to be the only players in the applications development field and if they are left their alone too long they will get too much of a head start ever to be caught. (And Perl seems to be available in the most recent snapshot) There is a fantastic opportunity here for the tcl community to show how the effectiveness of the language. In many cases tcl is faster than Python and Java, it is much simpler to write and debug programs and the GUI primitives are much better integrated into the system as a whole.

To be useful a handheld machine needs the typical PDA applications: diary, to-do list handler etc. etc. There are probably tcl applications out there that already do most of these things and they could be cleaned up and ported. (In fact, I am working on my to-do list manager Spike). A web browser would be nice: the tk html widget and the Http package give you a big step up here. And how about an ICQ client? An email reader? A Jabber client? Again applications that exist at least partially and which should move without much difficulty – as we all know, tcl is the easiest language to port between platforms.

There is already a work taking place to build a simple object database for the iPaq that could then be used as a PDA data repository. The developers are targeting Python applications but there is no reason why there should not be a tcl layer above it to provide the interface.

Tcl seems to me to be a natural match for handheld systems, but to reach a wider public the applications need to be there and they need to be written now before the other systems get a stronger foothold. All the basic tools and nuts and bolts are there they just need to be assembled, so let's start putting things together!

Acknowledgments

All the people working on and using tcl on the iPaq, particularly Steve Reddler, Rene Limberger and Paul Healy

Further Information

Information about the iPaq and Linux can be found in many places on the Internet, but the best to go for information is

<http://www.handhelds.org/>

where you will find everything that you may require.



12 ENIÄK – High-level construction of user interfaces

Kristoffer Lawson (<mailto:setok@fishpool.com>)
Fishpool Creations Ltd (<http://www.fishpool.com>)

ENIÄK — High-level construction of user interfaces*

KRISTOFFER LAWSON
FISHPOOL CREATIONS LTD
`mailto:setok@fishpool.com`

\$Date: 2001/05/10 19:14:58 \$

Abstract

ENIÄK is a high-level protocol, object system and related libraries for controlling and building user interfaces (UIs) to work with a large variety of display platforms, without change to the application. The goal of the project is to allow the application designer to build a UI structure on an abstract conceptual level which can be concretised visually by an independent display server.

The protocol is built so that a relatively small amount of active communication takes place between the application and the display server. Thus it works fairly well even on networks with higher latency. ENIÄK application designers are encouraged to think of the logical structure of the UI instead of the exact representation. This means that the same application can be used, without recompilation, with display servers that use f.ex. Tk, Windows, MacOS, curses, HTML/HTTP etc.

1 Overview

Traditional UI systems tend to describe an interface as a set of visual widgets and layout directives. Modern systems like Tk can make this process straightforward and easy. However, the application developer is still forced to lose focus from developing the actual application and has to use time for placing buttons, fixing frames and windows, working out good spacing and moving widgets around to get a nice look and feel.

In addition, with UI construction being so concerned with matters of appearance, it is not always easy to change an existing application to work in different display environments. Tk and Java's AWT/Swing do provide a look and feel which is specific to the display platform, but there is a limit to what the libraries can do to achieve a truly integrated UI. It is also impossible to use these for HTTP or text applications.

Another problem with traditional systems is that they do not work well when running from a remote machine. Even with the mechanisms provided by the X Window System it can be quite inefficient. An X application uses certain primitives to describe the layout of the interface but the X server is not given higher level understanding of the various elements. This kind of system works adequately for LANs but in a larger network can make redraws and simple interaction unusably slow.

ENIÄK is an attempt to make the life of application developers easier by resolving all of these problems in one large out-stretched sweep and, as with many good ideas in computing, originated from programmer laziness.

1.1 Goals

Unlike Tk, ENIÄK is not actually an API specification. Instead it defines a general protocol, object types (akin to widgets), attributes and some construction rules. It does not specify how

*For the 2nd European Tcl/Tk User Meeting.

collections of objects, and their attributes, will be rendered — only what their semantics are. In this way a wide variety of clients, libraries and display servers can be used to visualise the logical structure on a particular platform. For example, a client library is currently being built using Tcl, but another one could easily be built for use with C, Java or another script language like Perl or Python. Similarly the main display server currently being worked on uses the TclHttpd and HTML, but in the future it would be important to have servers that implement rendering with Tk, Gtk, Qt and even the terminal library curses, as well as Windows and Mac OS libraries. Here it is also good to point out that ENIÄK does not really compete with other UI systems but rather provides a standard layer on top of them so that applications do not have to be designed with a particular display in mind.

The long-term goal is that developers should think of their user interfaces in terms of abstract objects forming a tree structure. The design philosophy behind this is similar to L^AT_EX and XML, where documents are described structurally instead of using rich text. The types of objects used are also defined to describe what they do rather than what they look like. An example of this is that there are no such things as windows or frames in ENIÄK, at least not in the traditional sense. Both of these are essentially just ways of grouping objects together on different levels, so in ENIÄK group objects are used instead.

With display servers we are attempting to automate many of the layout issues that usually plague application developers. There is a wide range of layout patterns used for different kinds of applications. As users we immediately recognise these patterns and with a little experimentation we are able to use applications that we have never seen before — so long as they follow the semi-unwritten laws that describe how applications of that class should operate. The belief is that these patterns can be translated into algorithms which are able to build a layout based on only a logical description of the application.

A side effect of this design is that the same application can run, without modification or recompilation, with a variety of different display platforms. Simply by connecting to a different address, the same application can operate as either an X, Mac or web application.

At this point it is useful to take a look at how this fits in with the Model View Controller pattern. With MVC the application's internal structure and operation is separated from the external interface via a controller which handles events and controls the application model. When this model changes, observers or views are informed and they in turn change the interface presented to the user or other external system.

This pattern is not forced on the developer and the role of the controller, model and views are still aspects of the application and possibly the client library. What ENIÄK does allow in MVC is for the views to generate only structural information about the interface to the independent display server. Thus part of the work often left to the views can be fully automatised and no longer burdens the application developer or client library.

One final target for the project is that, despite the huge numbers of possibilities and greatly reduced burden on application developers, it should be as simple as possible to implement protocol parsers, to set up basic display servers and most importantly client libraries. So, when possible, the straightforward approach has been chosen over a more complex one.

2 Communication

The core of ENIÄK is the protocol used between the client, the display server and possibly an application server. It uses a stream of messages to control objects, their attributes and other aspects of the application. Initially it was thought that XML could be used for the protocol. However as it is for just a series of simple messages, XML seemed like overkill. The benefit of being able to use available parsers is lost, but as the message format is so simple anyway the loss is not great. XML will be used in the future for ENIÄK resource files.

The format of each message is:

```
ID COMMAND [PARAM1=VAL1 PARAM2=VAL2 ...]
```

ID is a unique identifier of an object in the application's object hierarchy. A **COMMAND** is specified in the complete ENIÄK specification and is basically a message to the object with the given identifier. The parameter-value pairs define parameters to the command. The parameters and values a message can contain are specific to the command. Standard commands, parameters and some values are given in upper case characters.

If a message was dealt with successfully an **OK** message is sent in response. If an error occurs at any time an **ERR** message can be sent. Thus every message is responded to with either an **OK** or an **ERR**. If the message handler wishes to notify of potential errors or risky commands it can do so with the **WRN** message, but these must be followed by an **OK** or an **ERR** message in response to the original message.

For example (the arrows show the direction of data flow and are not part of the message):

```
> 0 NEW TP=GRP ID=MainGroup
< 0 OK
> MainGroup NEW TP=BTN ID=OkButton NM=OK
< MainGroup OK
> MainGroup NEW TP=BTN ID=Cancel NM=Cancel
< MainGroup ERR ERR='Syntax error' DET='Missing '=''
```

Here the application successfully creates a new group under the root object. The root object is created automatically when an application connects to a display server and is given the identifier 0. After that the application creates an **OK** button and attempts to create a **Cancel** button, but for some reason the equal sign is missing between **ID** and **Cancel**. The display server reports this as a syntax error.

3 Objects

ENIÄK objects form a hierarchy which describes the structure of the user interface. Each object may have a variety of attributes which describe properties of the object. Each object can have any number of children and one parent. At the top of this hierarchy is the root object with attributes that usually affect the application as a whole. The parent of the root object is itself.

ENIÄK specifies many types of objects such as buttons, text, event detectors, groups etc. Instead of going to great measures to describe which types of objects can fit with other types of objects, a looser approach has been taken. Any type of object can be the child of any other type of object. It is then the job of the display server to sort out the mess and display something that reflects the hierarchy in the best possible way.

Each object can have any number of attributes. Some attributes have special meanings for all objects or certain object types. Upper case is used for standard attributes and mixed case for extensions.

Each object has a type, which is defined with the **TP** attribute. A type is actually a hierarchical string where sub-types are separated from super-types with a dot. One example of this is the event detection type (**EVNT**). While it is not possible to create an instance of an **EVNT** object, it has several sub-types, like activation detection (**EVNT.ACT**), that can be used. As with attributes, upper case is used for standard object types and mixed case for extensions. However, extensions cannot be made to top-level object types. This means an application and display server cannot use an object of the type **foo**, but **GRP.foo** is allowed. If a display server does not recognise the extension, it defaults to the first super-type that it understands. With **GRP.foo** this means that a display server will default to **GRP** if it does not understand the **foo** extension.

When defining layout algorithms the parent object is an important factor and the relation of the parent to the child is a "rules over" relationship. So the parent is considered to be the more important of the two, with wider consequences and a child, in a sense, belongs to its parent. How this is then interpreted depends on the display server and the types of object.

It should also be emphasised that similar object structures might be visualised in very different ways by the display server, depending on the context, object count etc. After all, the goal of the

project is to allow the developer to concentrate on structure and let the display server generate a suitable interface. A very simple example of this would be with selection lists. With a short list of options a drop down list might suffice, but if this list grows to contain even hundreds of options the display server should probably group these and generate pages for each group. It is not difficult to describe more complex cases, which may require quite a lot of work from the display server and its implementor. The point to remember is to allow the developer to focus on using structures without thinking of the result and to generate good-looking output whatever the input.

Some examples of object types:

BTN	Button. For setting up control points in the interface. With buttons the user can quickly access important features of the application.
EVNT	The super-type for objects detecting events.
GRP	Used to group objects together.
TXT	Text and editable text areas.

4 The application server



Figure 1: Normal application architecture.

In some situations the display server itself may want to start applications. This can be useful in desktop systems where double clicking on an icon results in communication with a display server, which then starts the appropriate application. This application will often then connect back to the display server to build the user interface. Another similar situation occurs with HTTP -based display servers where a listening display server translates certain URLs into requests to start up applications. The ENIÄK application server allow these scenarios.

In the simple client-server model, an application is started (f.ex. from a command shell) and is told to communicate with a specified display server. The application connects to this display server and describes the interface structure. This is then translated into visual form for the user. This is shown in figure 1.

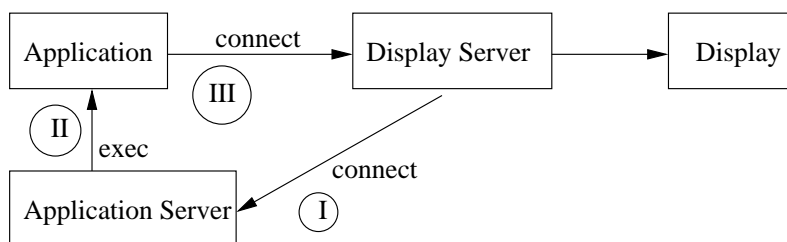


Figure 2: Tying an application server into an ENIÄK system.

Alternatively the display server can connect to an application server, if one is running, and instruct it to start an application. The application server can either start the application itself or redirect the request to another application server. When starting an application, the application server can then inform the application of the whereabouts of the display server and any other necessary information. After that, the application can connect to the display server just as with the simple model. This process is shown in figure 2.

5 A Tcl/XOTcl client library

The ENIÄK system itself does not specify anything about what client libraries should look like and how they should be implemented. However, it is impossible to imagine the system being in use without them. Developers would probably not be particularly charmed by the idea that they would have to read and write from sockets themselves!

As mentioned previously, work is commencing on a library based on Tcl and to be used with Tcl. It uses the extension XOTcl, which provides a clean, read/write introspective object oriented system. Many of the object types and features have been fairly directly mapped to equivalent XOTcl classes and methods allowing for a clean and straightforward way of creating ENIÄK applications from Tcl.

Here is a short (but complete) example of what code using the library might look like:

```
package require fishpool.eniak
namespace import eniak:*
proc changeText {btnOn evntOb} {
    global txt i root
    # Increase the counter and change the text in the text
    # object.
    incr i
    $txt text 'Count: $i'
    $root refresh
}
set i 0
# Setup connection with display server on localhost at
# port 24242. Call the application 'Example'.
set ds [EniakServer new localhost 24242 'Example']
# Get the automatically created root object.
set root [$ds getRoot]
# Create a text object as a child of the root object.
set txt [Text new $root]
# Set initial text.
$txt text 'Count: 0'
# Create a button (and activation event object, which the
# library creates automatically) that calls the changeText
# procedure when activated. It is a child of the text
# object.
set btn [Button new $txt 'Count' changeText]
# Changes are only guaranteed to happen after refresh is
# called on the parent.
$root refresh
vwait forever
```

So in 16 lines of actual code we can create a complete application that will run equally well in X as on the web (if the necessary display servers are available). It is not a terribly useful one, but even if we only had a web display server available, we can already see that this way is preferable to dealing directly with HTML and handling forms. The display server will do all of that automatically — and we can later run it as an X application!

6 A HTTP display server

The HTTP display server was created primarily because of actual need for such a product. Its implementation was bound to be less straight-forward than a Tk display server, for example,

because there would not be a persistent connection between the user and the display server. The ENIÄK clients (the applications) would connect to the display server and communicate using the ENIÄK protocol, and the users would have access to the applications using a web browser.

The obvious starting point for the HTTP display server was the TclHttpd web server, the functionality of which can be elegantly extended for tasks such as this, with just Tcl. The display server configuration includes the location of an application server and a list of applications that can be started. Individual URLs are assigned for starting each application. When the user visits an URL assigned for starting an application, this is what happens in more detail:

1. When the display server receives a HTTP request for the URL assigned for starting the application, it asks the application server to start an instance of the application.
2. The application connects to the display server.
3. The display server redirects the user's waiting web browser to an URL assigned for this particular instance of the application.
4. The display server receives the second HTTP request and returns the web client an HTML page representing the current state of the application.

The HTTP display server uses the Tcl event loop mechanism to handle data from the application server, the ENIÄK clients and the web clients asynchronously. For example between the possibly considerable delay between steps 1. and 2. in the example, data from other ENIÄK clients or requests from other web clients can be processed. Widgets such as buttons and text fields are represented by their HTML equivalents, and further interaction between the user and the application is done by handling the HTTP requests when the user submits the form — which often may only contain buttons and no actual input fields.

The display server code is a set of XOTcl classes. Some of the classes, like the base implementations of different types of ENIÄK objects, are common to all display server implementations, and provide a common framework with which it is easier to implement display servers for other environments.

The HTTP display server is still very much under construction, but it is already perfectly functional. The problems encountered so far are mostly related to the nature of the environment: the limitations of HTML and the lack of persistent connection to the user. The user can cancel the HTTP requests suddenly, for example, and making sure that the user is looking at what the application developer would want him to be looking at can be tricky.

7 Related work

Berlin is a project with many similar ideas as ENIÄK. It strives to provide a structured way to build user interfaces. It supports network transparency via the use of CORBA. In the future there are also plans to use different display servers for various platforms.

However, the focus is somewhat different. While the goal of ENIÄK is to integrate with existing environments (like the different available toolkits), the Berlin project is focused on implementing a new windowing system and a graphical display server. The ability to work with other environments in the future is more of a footnote than something that is core to Berlin. In comparison ENIÄK is only really a protocol and object specification and it is of high priority to work on a wide range of display servers and clients, as early on as possible. Berlin talks of structured graphics, while the term structured interface description is preferred in ENIÄK.

In addition, it is very easy to quickly understand the basics of ENIÄK and get a simple application running. In this way one is reminded of the simple efficiency that Tk programmers have grown used to.

However, despite the differences in focus, there may be areas where some collaboration can be done and so this should be looked at more carefully.

8 Conclusions

The HTTP display server, the XOTcl library and an application server are evolving continuously and have been used quite extensively for developing a couple of applications. A lot of effort has gone into making all the components work together — especially the display server and all the hidden logic involved. While this has taken time, developing the actual applications has been much quicker than what it would otherwise have been, plus there is the added bonus that they will integrate nicely with future display servers for other environments like X and Windows. This will become even simpler when the library is able to handle ENIÄK resource files. The work reflects the philosophy of any lazy programmer: do a bit more work in the beginning and much less in the future.

There is still a lot to be done. The system is already in use, but lags behind the specification and cannot neatly handle all object type combinations. In addition to that on-going effort, more display servers and client libraries for other languages and environments would result in a huge leap in the relevance of system for everyday life. Any community effort is extremely welcome. Please contact the author if you are willing to contribute in any small way.

References

- [1] John Ousterhout: An X11 Toolkit Based on the Tcl Language, 1991.
- [2] John Ousterhout: Tcl: An embeddable Command Language, 1990.
- [3] Kristoffer Lawson: ENIÄK v1.0, <http://dev.fishpool.fi/oss/eniak/doc/eniak.ps>.
- [4] ENIÄK homepage, <http://dev.fishpool.fi/oss/eniak/>
- [5] Brent Welch, The TclHttpd Web Server.
- [6] The GIMP Toolkit, <http://www.gtk.org/>.
- [7] Qt GUI Framework, <http://www.trolltech.com/products/qt/>.
- [8] L^AT_EX, A Document Preparation System, Leslie Lamport, 1994.
- [9] Extensible Markup Language (XML), <http://www.w3.org/XML/>.
- [10] G. Neumann, U. Zdun: XOTcl, an Object-Oriented Scripting Language. Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, USA, February, 2000.
- [11] XOTcl webpage, <http://www.xotcl.org/>.
- [12] Stefan Seefeld, Graydon Hoare: Berlin: A Structured User Interface.



mod dtcl web scripting with Tcl

David N. Welton (<mailto:davidw@apache.org>)

mod_dtcl

web scripting with Tcl

David N. Welton <davidw@apache.org>

May 21, 2001

What is it?

mod_dtcl integrates the Tcl scripting language with the Apache web server, so that it is possible to use Tcl for server-side web scripting. It is similar to PHP, in functionality.

Hello world

```
<h1>Some HTML</h1>
<?
hputs 'hello world'
?>

<? if { $foo == 1 } { ?>
More Html
...
<? } ?>
...
```

2

Advantages

- Avoid reinventing the wheel!
- Lots of existing Tcl code.
- Use code both for web projects and elsewhere.
- Lightweight.
- Easy to use.
- Fast!

3

One Language

With Tcl, you can take advantage of the existing mass of source code already available, and what's more, reuse the same code throughout your projects, involving the web or not.

4

Small and light

- `ls -l mod_dtcl.so` is about 45K on a Linux PPC system.
- `wc -l *.ch` is 3010 lines.
- Runtime Apache processes grow by about a meg with `mod_dtcl` , on a PPC Linux system.

5

Comparisons with other languages/systems

- CGI
- PHP
- mod_perl
- Zope
- Java
- Other Tcl solutions

6

CGI

CGI is a bit dated, and is inefficient, unless the process run is so heavy as to make the overhead of running it seem insignificant.

7

PHP

PHP is the example that `mod_dtcl` follows, but has a big limitation: it only runs on the web! It has a number of good features - it is small, light, reasonably fast, and is easy for non programmers to use. However, it has the disadvantage of being created for one and only one use, making it impossible or difficult to use elsewhere.

8

`mod_perl`

`mod_perl` takes a different approach - it lets you access the functionality of Apache with Perl. This makes it rather large, and at times difficult to install and use for simple things. In any case, we have `mod_tcl` (more on that later).

9

Zope

Zope is another very large and somewhat complex system. It may work well for larger sites that need some of the order and structure it imposes, but it is difficult to compare to `mod_dtcl`, which is aimed at a different target. It must be said, though, that Python is a nice language.

10

Server-side Java

Buzzwords galore!

Java isn't a higher level language. Strings are what "goes into" and "comes out of" web pages, so why not use a language adept at dealing with them? Java is harder to use, set up, and you may have licensing issues.

11

Other Tcl/Web platforms

AOLserver is a very nice package which demonstrates the power of Tcl. It is impressive to think that it (along with the proprietary StoryServer) were created in 1995, when most people were using Perl CGI's. It is a complete server from the ground up, and so doesn't ride on Apache. **NeoWebScript**, by Karl Lehenbauer, is similar to mod_dtcl, but does a few things differently. mod_dtcl was however, the **first** Free Software Tcl/Web system.

12

How does it work?

- Embedded Tcl interpreter.
- Pages run in separate namespaces.
- Cached bytecompiled scripts.
- Memory IO channel / cached output.
- HTML/Tcl integration.

13

Interpreter

One Tcl interpreter runs in each Apache process, and each .ttml file is run within a separate Tcl namespace. This lets .ttml files share variables, if necessary, but makes it difficult for pages to interfere with one another.

14

Cacheing

When a .ttml page is first loaded, the bytecompiled Tcl code is inserted into a cache, so that the next time the page is hit, it is not necessary to recompile the code, or even open the file again.

15

Memory IO Channel

When creating `mod_dtcl`, it was considered desirable to be able to run normal Tcl scripts with as little modifications as possible. In order to deal with standard output, `mod_dtcl` uses an in-memory file channel that then uses Apache's output functions. So `puts 'foobar'` works in `mod_dtcl`.

16

Mixing HTML and Tcl

Tcl and HTML can be mixed as much as is necessary. The `mod_dtcl` parser replaces sections of HTML as “`hputs`” statements. This lets you put bits of HTML in the middle of loops, in conditional statements, and so on.

17

Another Example

```
<?
set i 1
hputs "<table>"

while { $i <= 8 } {
  hputs "<tr>"
  for {set j 1} {$j <= 8} {incr j} {
    set num [ expr {$i * $j * 4 - 1} ]
    hputs [ format "<td bgcolor=%2x%2x%2x > $num $num $num </td>" \
      $num $num $num ]
  }
  incr i
  hputs "</tr>"
}

hputs "</table>"
?>
```

18

Produces...

1 1 1	7 7 7	11 11 11	15 15 15	19 19 19	23 23 23	27 27 27	31 31 31
7 7 7	15 15 15	23 23 23	31 31 31	39 39 39	47 47 47	55 55 55	63 63 63
11 11 11	23 23 23	35 35 35	47 47 47	59 59 59	71 71 71	83 83 83	95 95 95
15 15 15	31 31 31	47 47 47	63 63 63	79 79 79	95 95 95	111 111 111	127 127 127
19 19 19	39 39 39	59 59 59	79 79 79	99 99 99	119 119 119	139 139 139	159 159 159
23 23 23	47 47 47	71 71 71	95 95 95	119 119 119	143 143 143	167 167 167	191 191 191
27 27 27	55 55 55	83 83 83	111 111 111	139 139 139	167 167 167	195 195 195	223 223 223
31 31 31	63 63 63	95 95 95	127 127 127	159 159 159	191 191 191	223 223 223	255 255 255

19

The Future

- mod_tcl.
- Useful functions (library).
- Safe mode.
- Nice logo!

20

mod_tcl

mod_tcl is the Tcl equivalent of mod_perl - it gives us access to the Apache API through Tcl. It is currently available for Apache 2.0.

21

Library functions

Currently a problem with Tcl in general - lots of code out there, but it's poorly organized and there is no quality control.

22

Safe mode

It hasn't been requested much, but it would be interesting to take advantage of Tcl's safe interpreters to make 'safe' pages of some kind.

23

Looking for a Logo

If anyone with an artistic bent has some ideas, they are more than welcome:-)

24

Conclusion

Tcl is an excellent language for the web. As a generalized, embeddable “command language”, it is a natural for the web, giving the user a powerful language with years of history and many capabilities, in an environment tuned for use with the web.

25

Contact Information

Web site: <http://tcl.apache.org/>

Email: davidw@apache.org