Everything You Always Wante
to Know about TCL

A – very brief(!) – introduction into Tcl as

Tool Command Language **and**

General Scripting Language

Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
http://tbfe.de

FPGA-Kongress 2016
Sprachen | Languages
2016-07-12 | 14:15 … 15:4

*: You may download this presentation from the Author's Internet Site for any use in complia Creative Commons BY-SA License. As it has been created using the free HTML4-Tool Remar is written using the Markdown-Syntax, so you may even enhance the purely electronic form with your own annotations, only by means of any ordinary text editor. Just hit the viewing this in an internet browser, and follow instructions.

Styled with styling.css by Martin Weitzel

Agenda

PART 1

- Tcl's General Design
- Tcl's Minimal Syntax
- Tcl's Standard Library

PART 2

- General Tcl Scripting
- Event-Driven Design
- Code-Walking a Real Projec

Also Available as BONUS TRACK

- Using Tcl in Vivado

You are welcome to interrupt the speaker with questions* and – es during the live examples and code-walks – propose to try small ch

*: Your questions will be answered during the presentation to the best of the speaker's abilitie in private communication after the presentation.

Tcl's General Design

The general design of Tcl combines:

- A mostly trivial syntax
    - supporting some basic data structures
- A standard library providing
    - constructions for flow control
    - some more data structures miscellaneous utilities*
    - introspection and hooks for debugging
- A table-lookup mechanism for dispatching commands
    - **which may be extended by tools** (e.g. Vivado)

Tcl's Minimal Syntax

Understanding *Tcl's Syntax* actually is:[*]

- Understanding
  - Command-Separation and
  - Word-Separation
- Understanding various Substitutions:
  - Variable names for the stored value
  - Subroutines/Commands for what they return
  - Escape-Sequences for Non-Printing Characters
- Understanding various forms of *Quoting*
  - Backslash Quoting
  - Double_Quote_Quoting
  - Curly Brace Quoting

[*]: To quote John Ousterhout who designed Tcl in the late 80s: *Many problems beginners have from the fact they assume a more complex syntax as their actually is.*

Tcl's `eval` command

This is the command you use

- *implicitly* all the time but

- *explicitly* in **very rare** cases only

It causes the *Tcl Syntax Parser* to

- look at any character string as Tcl command,
- applying word- and command separation,
- plus variable and return value substitution,
- while honoring all quoting,
- ending with looking up the first word as command,
- **– finally executing it –**
- handing over all the remaining words as arguments.

> In other words: it does what happens all the time when Tcl execu
> text file as script of commands.

Tcl's `expr` command

This is the command you

- **you need to use explicitly** …
- … in a context that needs to apply
  - basic logic (and, or, not) or
  - arithmetic operations (plus, minus, …) and
  - mathematical functions (square root, sine, cosine, …)

Command Separation

Tcl separates commands at the following boundaries:

- Newline(s)*
- Semicolons

Note that:

- Command separators are looked for **prior** to any substitution
- So especially newlines "generated" via the escape sequence **not** command separators

*: I.e. ASCII/Unicode Code Point 10 / 0x0A.

Word Separation

Tcl separates words **within commands** at the following boundarie

- Space characters
- Horizontal tabulators[*]
- Sequences of the above

Note that:

- Word separators are looked for **prior** to any substitution
- So especially tabulators "generated" via the escape sequenc are **not** word separators

[*]: I.e. ASCII/Unicode Code Point 9 / 0x09.

Substituting Variable Content

Tcl substitutes a variable name for the content of that variable

- When the name[*] of a variable is preceded by `$`
  - and the variable exists (i.e. has a value assigned to it)
  - **otherwise it is an error**

[*]: The spelling of variable names in Tcl are a (slight) super-set of what is valid in most other languages. By enclosing the variable name **after** the `$`-sign into curly braces the rules more "relaxed", so that nearly **anything** will be acceptable as variable name – though re be hurt if this freedom is exploited too frequently.

Substituting Subroutine Return Values

Tcl substitutes the return value of a subroutine

- If the complete command, i.e.
  - command name and all arguments following
  - are enclosed in square brackets
- For the part inside square brackets
  - a recursive syntax analysis is started
  - which may in turn use square brackets
  - leading to nested command substitutions[*]

> Note that this applies as well to **all built-in commands** of Tcl
> (additional) sub-routines defined via the `proc` commands.

[*]: Though technically many levels are possible, for readability any nesting should be kept r
storing return values required as arguments (to other commands in temporary variables w
name".

Escape Sequences for Non-Printing Characte

This is a Tcl feature taken from the C programming language:

- `\a` → Audible Bell Character (ASCII/Unicode Code Point 7 / 0x0
- `\b` → Backspace (ASCII/Unicode Code Point 8 / 0x08)
- `\t` → Horizontal Tab (ASCII/Unicode Code Point 9 / 0x09)
- `\n` → Newline Tab (ASCII/Unicode Code Point 10 / 0x0a)
- `\v` → Vertical Tab (ASCII/Unicode Code Point 11 / 0x0B)
- `\r` → Carriage Return (ASCII/Unicode Code Point 12 / 0x0C)

Octal and hexadecimal notations are supported too, like in C

Backslash Quoting

A backslash may preceded any character.

- If this is **not** an escape sequence for a non-printing character
- … the character directly following is taken verbatim. e.g.
  - the word separators are taken as a verbatim space or hori tabulator (character)
  - the variable substitution request `$` is a verbatim dollar usi
  - the command substitution request `[` and `]` are verbatim sq brackets using `\[` and `\]`
  - the command separator `;` is a verbatim semicolon using `\`
  - **to obtain a single backslash it needs to be written a**

> Note that a backslash at the end of a line is a special case:
> Together the newline character following **and** all white space at t
> start of the next line it is **replaced with a single space charac**

Double Quote Quoting

Any sequence of characters my be enclosed in double quotes and

- command and word separation will **not take place**
- but anything else **will** work as if it were unquoted:
    - $ requests *Variable → Value* substitution
    - […] requests *Command → Return Value* substitution
    - \ will
        - **either** quote one of the above
        - **or** work as escape sequence for non-printing characte

Note that with respect to quoting with double quotes a backslash

- **outside** turns an (initial) `\"` into a verbatim double quote;
- **inside** turns `\"` into representing a (contained) double quote

Curly Brace Quoting

Any sequence of characters may be enclosed in a pair of curly bra
inside

- any characters contained are taken verbatim
- up to the (matching) closing curly brace

Note that

- the **matching** curly brace is determined by
  - counting opening and closing braces
  - but **not** removing them;
- a **contained** curly brace is
  - exempt from being counted
  - but the backslash is **not** removed.

Tcl Syntax Summary

Now you have learned (nearly) all[*] of Tcl's trivial syntax.

> Be sure to remember what John Ousterhout said and
> **do not assume a more complex syntax as there actually**

[*]: Probably 99,9% of what you need in any Tcl script you'll ever write.

Tcl's Standard Library

The speaker will now continue with live examples.

During a self-study you may want to look-up command documenta
linked below to see typical examples.

Branches and Loops:

- `if`, `switch`, `while`, `for`, `foreach`,
- `break`, `continue`

Sub-Routines and Error Handling:

- `proc`, `return`, `error`, `catch`

Introspection and Debugging:

- `info`, `trace`, `rename`

Data Handling:

- plain variables,
- arrays,
- lists, and
- dictionaries

Classic Library Utilities:

- much more* is provide
  parts of it will be show

*: As string handling including regular expressions, input ([by line] or [by number of characte
classic file interface with open and close) and TCP/IP network sockets too , accessing the fi
listing directories and operations on whole files), time and date etc. ... and support for a
architecture with an event driven design

Tcl for General Scripting

As general scripting language Tcl has its Pro's and Con's:*

- Advantages are:
    - Open Source (continuation does not depend on a third par
    - Mature, stable and always striving for backward compatib
    - Established community of dedicated users
    - Proven use in many serious projects (since ~25 years)
    - Efficient and small memory footprint
    - Extensible to GUI-Programming with Tk
    - **Known from FPGA Scripting Tools**
- Disadvantages are:
    - Not much hype anymore (nowadays)
    - Might be considered "out-dated" and therefore …
    - … not be attractive to many software developers
    - Relatively small user community
    - Nifty "modern" GUI-Controls missing

*: Compared to "more modern" alternatives.

Tcl and Event Driven Design

Tcl lends itself well to event driven software designs.

- An event-driven architecture
    - consists of may small handlers
    - to which a central dispatches events
    - is preferable anyway for GU-programming
- It usally depends on the posibility for an application …
    - … to send events to itself
    - … either with or without delay
- The design challenge is to keep handlers small

Pro's and Con's of the Event-Driven Approac

Advantages include:

- Handlers run single-threaded to their end
    - No need for mutexes or any other thread synchronisation
    - Therefore no worries
        - accidental data inconsistencies,
        - about race conditions,
        - or deadlocks

Disadvantages include:

- Forcibly breaking long handlers may not feel "natural"
- Will not easily scale to multi-core hardware

Tcl Commands for Event-Driven Architecture

The following commands are essential for event-driven Tcl prograr

- `after` – execute some handler deferred or continue via event-l
- `fileevent` – execute some handler when there is
  - data available **when reading** from a file or socket
  - space available* **when writing** to a file or socket
- `socket -server` – provide a "half-open" TCP/IP connection to ac connection requests from clients in a server application

> There is also a command `update` to enter the event-loop recursive using it **is not recommended because handlers have to con to be called re-entrant.**

*: Though most Tcl programs do not care about it, filling-up output buffers *might* lead to a being blocked in an event handler an become unresponsive.

Code-Walking a Real Project

The speaker will take you on a code-walk through a real proje

The code was written some years ago to interface with measurem data sent from a digital multimeter through a serial interface.

You may feel that "some years" is a bit of understatement:

Actually this multimeter was bought by the author about 15 years and the code shown was written around that time. The interesting it still works essentially unchangend on any system with a Tcl inter that also offers the Tk extension for GUI programming.*

- Whenever it was "ported" to a different host the only change **adapt the device name of the serial port**.
- Whenever it had to be "ported" to more recent version of Tcl/T **no changes at all had to be made**.

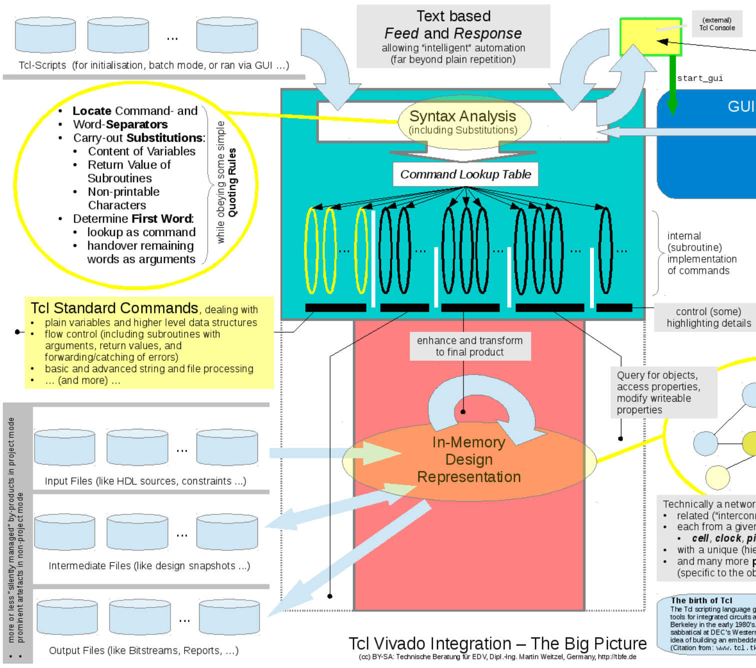
*: Though it hardly might make sense in practice, the code would – at least in principle – a *Embedded Tcl Interpreters* like the ones integrated in design tools for scripting. The only the event-loop of Tcl/Tk might not smoothly interface with an event-driven GUI. Vivado G uses its own event-loop which has no provisions to merge-in other event-loops.

Using Tcl in Vivado

- Tcl and Vivado – The Big Picture
    - Non-Project- vs. Project Mode
    - Understanding Project Mode
    - Interaction of Design Model and Tcl
- Vivado Command Conventions
    - Tcl vs. Vivado Commands
    - Necessity of Quoting
    - Storing Commands in Variables
- Understanding the Design Model
    - Basics of Design Navigation
    - Accessing Object Properties

Tcl and Vivado – The Big Picture

- Being **A *Tool Command Language* by Design** …
- … Tcl lends itself perfectly as scripting language for Vivado …
- … though the added commands not always follow conventions and style of (native) Tcl



Tcl-Scripts (for initialisation, batch mode, or ran via GUI …)

Text based
Feed and *Response*
allowing "intelligent" automation
(far beyond plain repetition)

(external)
Tcl Console

`start_gui`

GUI

- **Locate** Command- and
- Word-**Separators**
- Carry-out **Substitutions**:
  - Content of Variables
  - Return Value of Subroutines
  - Non-printable Characters
- Determine **First Word**:
  - lookup as command
  - handover remaining words as arguments

while obeying some simple **Quoting Rules**

Syntax Analysis
(including Substitutions)

Command Lookup Table

internal (subroutine) implementation of commands

control (some) highlighting details

Tcl Standard Commands, dealing with
- plain variables and higher level data structures
- flow control (including subroutines with arguments, return values, and forwarding/catching of errors)
- basic and advanced string and file processing
- … (and more) …

enhance and transform to final product

Query for objects, access properties, modify writeable properties

In-Memory Design Representation

Input Files (like HDL sources, constraints ...)

Intermediate Files (like design snapshots ...)

Output Files (like Bitstreams, Reports, …)

- more or less "silently managed" by-products in project mode
- prominent artefacts in non-project mode

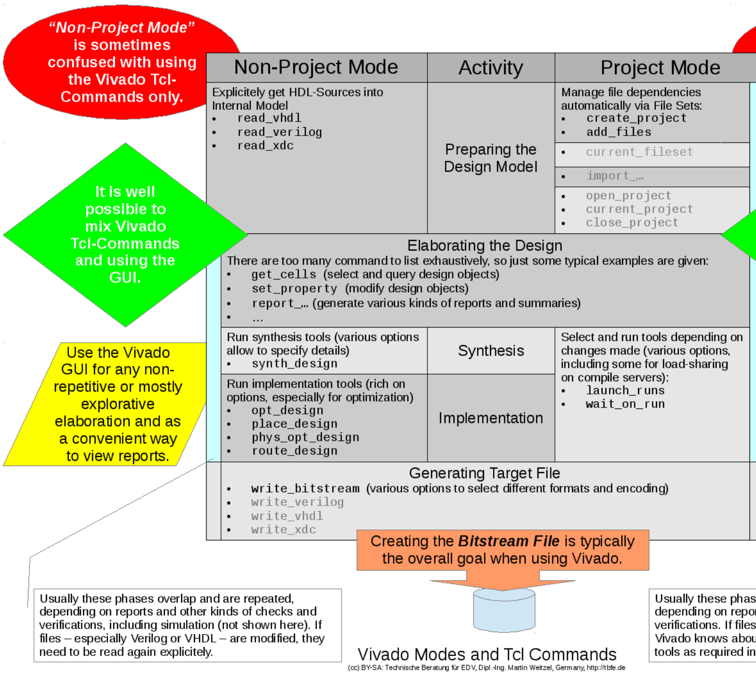
Technically a networ
- related ("intercon
- each from a give
  - ***cell***, ***clock***, ***pi***
- with a unique (hie
- and many more **p**
  (specific to the ob

The birth of Tcl
The Tcl scripting language g
tools for integrated circuits a
Berkeley in the early 1980's.
sabbatical at DEC's Western
idea of building an embedda
(Citation from: `www.tcl.tk`

Tcl Vivado Integration – The Big Picture
(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

Non-Project vs. Project Mode

- *Non-Project Mode* and *Project Mode* are two ways to use Vivado during design elaboration …
- … but the difference is **not** how much of the work is done via GUI and how much with Tcl



"Non-Project Mode" **is sometimes confused with using the Vivado Tcl-Commands only.**

It is well possible to mix Vivado Tcl-Commands and using the GUI.

Use the Vivado GUI for any non-repetitive or mostly explorative elaboration and as a convenient way to view reports.

Non-Project Mode	Activity	Project Mode
Explicitely get HDL-Sources into Internal Model • `read_vhdl` • `read_verilog` • `read_xdc`	Preparing the Design Model	Manage file dependencies automatically via File Sets: • `create_project` • `add_files` • `current_fileset` • `import_…` • `open_project` • `current_project` • `close_project`
Elaborating the Design There are too many command to list exhaustively, so just some typical examples are given: • `get_cells` (select and query design objects) • `set_property` (modify design objects) • `report_…` (generate various kinds of reports and summaries) • …		
Run synthesis tools (various options allow to specify details) • `synth_design`	Synthesis	Select and run tools depending on changes made (various options, including some for load-sharing on compile servers): • `launch_runs` • `wait_on_run`
Run implementation tools (rich on options, especially for optimization) • `opt_design` • `place_design` • `phys_opt_design` • `route_design`	Implementation	
Generating Target File • `write_bitstream` (various options to select different formats and encoding) • `write_verilog` • `write_vhdl` • `write_xdc`		

Creating the ***Bitstream File*** is typically the overall goal when using Vivado.

Usually these phases overlap and are repeated, depending on reports and other kinds of checks and verifications, including simulation (not shown here). If files – especially Verilog or VHDL – are modified, they need to be read again explicitely.

Usually these phas
depending on repo
verifications. If files
Vivado knows abou
tools as required in

Vivado Modes and Tcl Commands

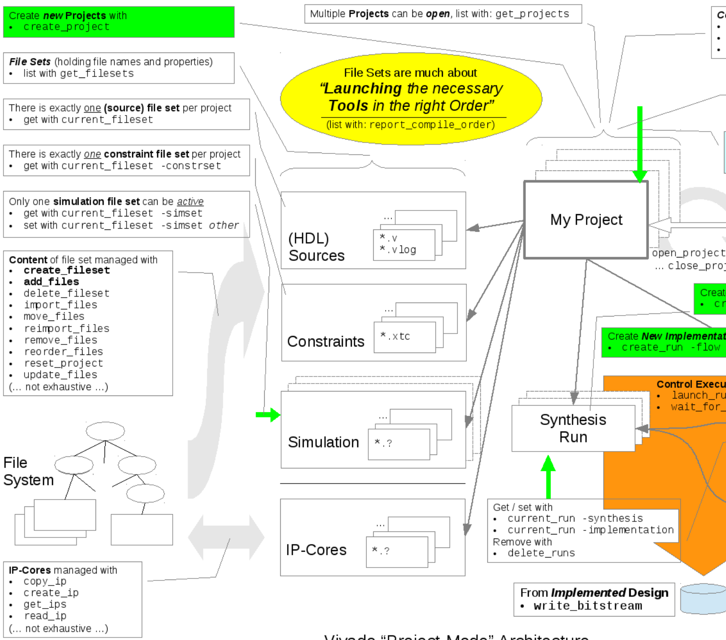
(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

Understanding Project Mode

- Relationships in *Project Mode* need some more explanation

- Basically it automates managing dependencies between files while minimising tool use

- Also options for various optimisation runs are specified in a packaged form, called *Strategies*



Create ***new* Projects** with
- `create_project`

File Sets (holding file names and properties)
- list with `get_filesets`

There is exactly <u>one</u> **(source) file set** per project
- get with `current_fileset`

There is exactly *<u>one</u>* **constraint file set** per project
- get with `current_fileset -constrset`

Only one **simulation file set** can be *<u>active</u>*
- get with `current_fileset -simset`
- set with `current_fileset -simset` *`other`*

Content of file set managed with
- **`create_fileset`**
- **`add_files`**
- `delete_fileset`
- `import_files`
- `move_files`
- `reimport_files`
- `remove_files`
- `reorder_files`
- `reset_project`
- `update_files`

(… not exhaustive …)

File System

IP-Cores managed with
- `copy_ip`
- `create_ip`
- `get_ips`
- `read_ip`

(… not exhaustive …)

Multiple **Projects** can be ***open***, list with: `get_projects`

File Sets are much about
*“**Launching** the necessary **Tools** in the right Order”*
(list with: `report_compile_order`)

(HDL) Sources — `*.v` `*.vlog` …

Constraints — `*.xtc` …

Simulation — `*.?` …

IP-Cores — `*.?` …

My Project

`open_project`
… `close_proj`

Creat
- `cr`

Create ***New Implementat***
- `create_run -flow`

Control Execu
- `launch_ru`
- `wait_for_`

Synthesis Run

Get / set with
- `current_run -synthesis`
- `current_run -implementation`

Remove with
- `delete_runs`

From ***Implemented* Design**
- **`write_bitstream`**

Vivado “Project-Mode” Architecture

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

Vivado Command Conventions

As already has been mentioned Tcl commands

- evaluate their arguments themselves (each)
- hence achieve "uniformity" by following conventions only
- where *Tcl Style* and *Vivado Style* slightly differs

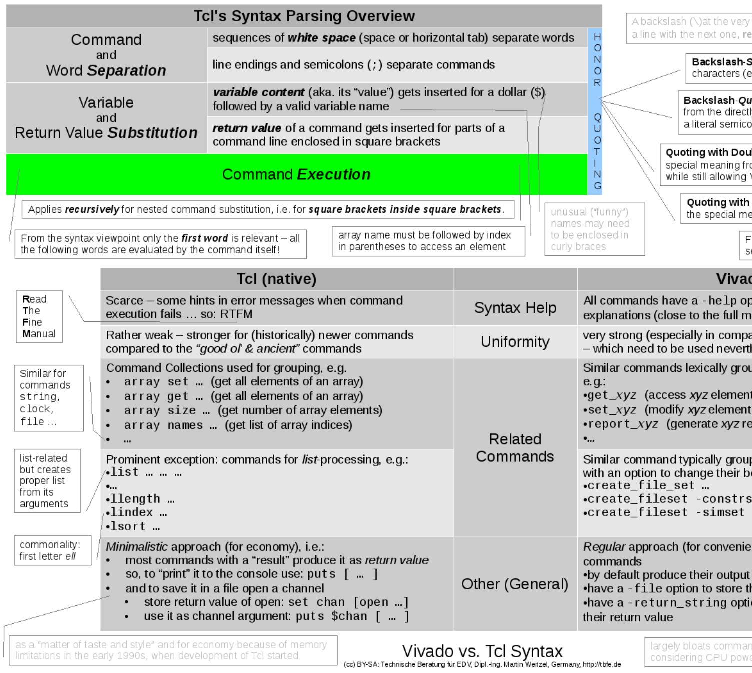
Differences are not quite from separate worlds but visible eno

Therefore it pays to be aware of a command's origin – Tcl or Vivad

[*]: ... somewhat like being aware whether driving a gear-shifting car or a car with automatic t Tcl is more on the gear-shifting side, Vivado more on the other (author's advice and persona The reasons for the differences are manifold and partially speculative: (1) Tcl itself is not c if made from one piece" and therefore not quite consistent. (2) There may have been the more "user friendly" in Vivado. (3) Scarce resources originally guiding some Tcl design de longer an issue today.

Tcl vs. Vivado Commands

- **Common** to Tcl and Vivado is Syntax Analysis only
- Beyond that slightly different styles become apparent



Tcl's Syntax Parsing Overview	
Command and Word *Separation*	sequences of ***white space*** (space or horizontal tab) separate words
	line endings and semicolons (`;`) separate commands
Variable and Return Value *Substitution*	***variable content*** (aka. its "value") gets inserted for a dollar ($) followed by a valid variable name
	return value of a command gets inserted for parts of a command line enclosed in square brackets
Command ***Execution***	

A backslash (\\)at the very
a line with the next one, re

Backslash-S
characters (e

Backslash-*Qu*
from the directl
a literal semico

Quoting with Dou
special meaning fr
while still allowing

Quoting with
the special me

F
se

Applies ***recursively*** for nested command substitution, i.e. for ***square brackets inside square brackets***.

From the syntax viewpoint only the ***first word*** is relevant – all the following words are evaluated by the command itself!

array name must be followed by index in parentheses to access an element

unusual ("funny") names may need to be enclosed in curly braces

Read **T**he **F**ine **M**anual

Similar for commands `string`, `clock`, `file` …

list-related but creates proper list from its arguments

commonality: first letter *ell*

Tcl (native)		Vivad
Scarce – some hints in error messages when command execution fails … so: RTFM	Syntax Help	All commands have a `-help` op explanations (close to the full m
Rather weak – stronger for (historically) newer commands compared to the *"good ol' & ancient"* commands	Uniformity	very strong (especially in compa – which need to be used neverth
Command Collections used for grouping, e.g. • `array set …` (get all elements of an array) • `array get …` (get all elements of an array) • `array size …` (get number of array elements) • `array names …` (get list of array indices) • …	Related Commands	Similar commands lexically grou e.g.: •`get_xyz` (access *xyz* elemen •`set_xyz` (modify *xyz* element •`report_xyz` (generate *xyz* re •…
Prominent exception: commands for *list*-processing, e.g.: •`list … … …` •… •`llength …` •`lindex …` •`lsort …`		Similar command typically group with an option to change their be •`create_file_set …` •`create_fileset -constrs` •`create_fileset -simset`
Minimalistic approach (for economy), i.e.: • most commands with a "result" produce it as *return value* • so, to "print" it to the console use: `puts [ … ]` • and to save it in a file open a channel • store return value of open: `set chan [open …]` • use it as channel argument: `puts $chan [ … ]`	Other (General)	*Regular* approach (for convenie commands •by default produce their output •have a `-file` option to store th •have a `-return_string` optio their return value

as a "matter of taste and style" and for economy because of memory limitations in the early 1990s, when development of Tcl started

Vivado vs. Tcl Syntax

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

largely bloats comman
considering CPU powe

Understanding the Design Model

Prior to navigating* within the design using Tcl commands there ne
be a basic understanding the model itself.

(i) For more information refer to:
XILINX UG894 **Using Tcl Scripting** – for an Introductior
XILINX UG835 **Vivado Tcl Commands** – Complete Refe

*: Navigating the design model is usually the first step to select one or more objects, which are accessed or modified.

Basics of Design Navigation

Navigating to objects in the design model is similar to navigating t
in a directory tree.

- There is a *Top Level Object* …
  - (much like the root directory of a file system)
- … which may also be changed
  - (much like the current working directory)

(i) UG 894 → **Tcl Scripting in Vivado**
→ Accessing Design Objects
→ Getting By Name – Traversing the Design Hierarchy

Understanding Object Relations

Objects are inter-related via connections*

- which may also be used as a base for navigation
- but first the relationships need to be understood

(i) UG894 → **Tcl Scripting in Vivado**
→ Accessing Design Objects
→ Getting Objects by Relationship

*: For basic and even for moderately ambitious tasks it is fully sufficient to grasp the main while ignoring the details. For the full picture see: *UG 835 → Ch. 1: Introduction → First Cla and Relationships → Object Relationships*

Navigating via Relations

The general form is this:[*]

- `get_`*`kind`* `-of_objects` *`which`*, where
  - *`kind`* depends on the type of objects **to be** looked-up, and
  - *`which`* is the type of objects **from which** the relation origin

Not all combinations of *`kind`* and *`which`* are valid!

Get accustomed to look-up proper usage in XILINX UG835 or use t command with the `-help` option interactively.

[*]: Note that the syntax chosen seems to strive for being readable in natural language:

```
get_pins -of_objects [get_nets -hier]
```

Filtering on Selection

Many commands selecting objects have a `-filter` option

- The required syntax deviates somewhat from Tcl style
- Often it makes sense to put the whole selection in curly braces

Be aware no variable substitution takes place in curly bra

*: Be aware of backslash substitutions (for `\n`, `\t`, etc.) if you use double quotes instead and word (and command) separation if you use no quoting at all.

Accessing Object Properties

Design objects generally have properties*

- Some properties are common to all objects
- Others vary with the type of object
- (also filtering is based on properties)

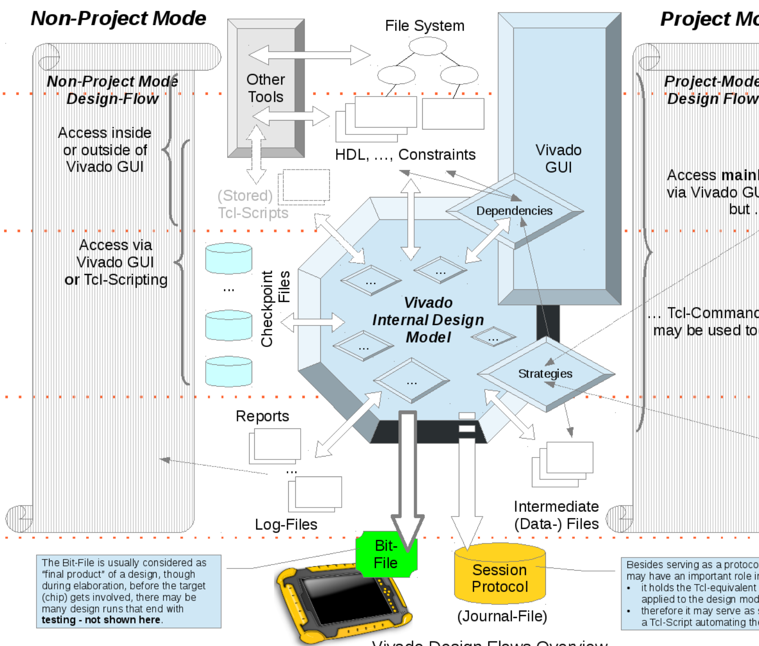
(i) For more information see:
XILINX UG835 → Ch. 1: Introduction
→ First Class Tcl Objects and Relationships
→ Object Properties

- XILINX UG835 → Ch. 3: Tcl Commands Listed Alphabetically →
  - `get_property`
  - `list_property`
  - `report_property`
  - … (and others) …

*: As this presentation could never be exhaustive without becoming a reference manual, no
that direction is made at all: get accustomed to look-up relevant information in XILINX UG8

From Design Model to Bitstream

- The typical final goal of any design is the *Bitstream File*
- Before this can happen the design usually needs some elaboration
- (Thorough testing not shown here – though highly to recommend before generating and using a bitstream file)


Non-Project Mode
Project Mo
Non-Project Mode Design-Flow
Access inside or outside of Vivado GUI
Access via Vivado GUI or Tcl-Scripting
Other Tools
File System
HDL, …, Constraints
(Stored) Tcl-Scripts
Vivado GUI
Dependencies
Checkpoint Files
…
Vivado Internal Design Model
Strategies
Reports
Log-Files
Intermediate (Data-) Files
Bit-File
Session Protocol
(Journal-File)
Project-Mode Design Flow
Access mainl via Vivado GU but …
… Tcl-Command may be used to

The Bit-File is usually considered as "final product" of a design, though during elaboration, before the target (chip) gets involved, there may be many design runs that end with **testing - not shown here**.

Besides serving as a protoco may have an important role i

- it holds the Tcl-equivalent applied to the design mod
- therefore it may serve as a Tcl-Script automating th

Vivado Design Flows Overview

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

*: Of course there are many reasons why a bitstream file might be never produced. E.g. a turn out to be inappropriate during elaboration or testing and is completely overturned. Or just to test tools Vivado uses internally. Finally, the bitstream files eventually produce workshop are also not of much interest … (and will probably never be loaded to a concrete

Non-Project Mode Outline

In *Non-Project Mode*

- Files (Verilog/VHDL, Constraints, Simulation) need to explicitly
- Toos (Synthesis, Implementation/Optimizations) need to explic

(i) For an outline of a session in non-project mode see:
XILINX UG888 → Lab 1: Using the Non-Project Design Flo

Project Mode Outline

In *Project Mode*

- Vivado manages dependencies via *File Sets*
- Arranges automatically for the adequate tools to run
- Provides optimisation via *Strategies*

> (i) For an outline of a session in project mode see:
> XILINX UG888 → Lab 2: Using the Project Design Flow

And More – If Time Allows

So you name it …

… otherwise:

Thanks for Listening!