Anhang C

Komponenten des Testsystems

Dieser Anhang enthält die Dokumentation der einzelnen Perl-Module und -Skripte, die im Testsystem verwendet werden. Die Dokumentation dieser Module steht direkt in den Quelltextdateien. Die Texte dieses Anhangs sind mit den Standard-Perl-Werkzeugen (`pod2latext` bzw. `pod2html`) aus diesen Quelltexten generiert worden.

Inhaltsverzeichnis

C.1 Perl Module CT.pm

Overview

This Perl module implements a model of a so called *classification tree*, implements the import of cte-files — the documents created by the *classification tree editor*, a mechanism to evaluate attributes of classification tree elements and an experimental export to a XML-document for documenting purposes.

The intended use is to read and evaluate a given classification tree. Manipulating the tree is not implemented completely.

The classes of the module

The Module contains at first classes for the elements of the *Classification Tree*, at second classes for the *Test Case Specification* and finally it contains a managing class for the classification tree itself and a helper class for handling attributes and their evaluation.

Class CT

The Classification Tree.

Creation

```
my $ct = new CT;
$ct->read_from_cte_file($cte_file_name);
```

Methods and Properties

- **constructor method** `new`
  Creates a new CT object.

  ```
  my $ct = new CT;
  ```

- **properties** `element`, `elements`
  `element` returns a tree node given by the CTE-internal ID. `elements` returns a reference to an array containing all tree nodes indexed by their ID.

  ```
  $el = $ct->element($cte_id);

  $aref = $ct->elements;
  foreach $el (@{$aref}) {
          do_anything_with($el);
  }
  ```

- **property** `frame`
  `frame` returns the CT::Frame object given by its CTE-internal ID.

  ```
  my $frame = $ct->frame($frame_id);
  ```

- **properties** `testobj_by_ctid`, `..._by_namepatt`, `..._by_test_id`
  All return a test object, i.e. a `CTE::TestCase` or a `CTE::TestSequence`. If they didn't find an appropriate test object they return `undef`.

  `testobj_by_ctid` selects the test object by its CTE defined ID.

  `testobj_by_namepatt` selects the test object with a name property which matches a given pattern.

  `testobj_by_test_id` returns a test object selected by its attribute named `test_id`.

  ```
  $to = $ct->testobj_by_ctid(17);
  $to = $ct->testobj_by_namepatt('^TC1000:');
  $to = $ct->testobj_by_test_id('TC1000');
  ```

- **property** `root`
  Returns the root node of the classification tree.

- **method** `read_from_cte_file`
  Reads a classification tree from a CTE file.

  Note: Currently it does not import all information available in a CTE file. Above all, layout information is completely ignored and some information introduced by newer versions of CTE to.

```
$ct->read_from_cte_file('d:\any_path\any_file.cte');
```

- **method** `print_to_xml`
  `print_to_xml` writes the classification tree and the test specifications to a filehandle as XML data. This XML currently is thought as documentation. Actually, not all information is written out.

  The method has two optional parameters: one that gives a path used as system identifier in the document type declaration, and 2nd a path used to link the document with an XSL stylesheet.

```
open(XML, ">a_file_name") || die qq(can't open);
$ct->print_to_xml(*XML);
$ct->print_to_xml(*XML, '../xml/testspec.dtd',
                        '../xml/testspec.xsl');
```

  The method takes as fourth argument, specifying the extent of the XML output. It defaults to 0. If it is greater than 0 for each test step or test case a list of all evaluated attributes is exported to. This may create quite larger files and takes a little more time but may be useful to find bugs in the test specifications.

Class CTAttributes

Methods and Properties

- **method** `make`
  Makes (blesses) a normal hash into a `CTAttributes` object.

```
$ahashref = { ... };
$atts = CTAttributes->make($ahashref);
$atts->evaluate(...);
```

- **method** `add`
  Adds all given hashes (given as references) to the attribute hash. Attributes in the new hashes override the previous settings.

```
$ahashref1 = { "a1" => "v1", "a2" => "v2" };
$ahashref2 = { "a3" => "v3", "a2" => "v4" };
$atts = CTAttributes->make($ahashref1);
$atts->add($ahashref2);
```

  sets `$atts` to:

```
{ "a1" => "v1", "a2" => "v4", "a3" => "v3" }
```

- **method** `set_context`
  Sets the so called *evaluation context*. The given object must implement the `predefined_var`-property. That is used by the `get_var`-method to find not directly defined attributes.

- **method** `get_var`
  Returns the value of an attribute. If the attribute is not defined in the underlying hash the so called *evaluation context* is requested for the value using its `predefined_var`-property. It returns `undef` if no such attribute can be found.

- **method** `set_var`
  Set the value of an attribute.

- **method** `del_var`
  Deletes an attribute from the hash.

- **method** `ren_var`
  Renames an attribute.

```
$atts->{'att1'} = 'value';
$atts->ren_var('att1', 'att2');
$v1 = $atts->{'att1'};
$v2 = $atts->{'att2'};
```

  sets `$v1` to `undef` and `$v2` to `'value'`.

- **method** `get_names`
  Returns an array of all variable names

- **method** `expand`
  Expands all attribute values and attribute names using the `expand_string`-method. The 1st parameter is the prefix that marks Variables.

  Note: Both, variable values and names are expanded, but with two differences: 1. variables in names are only expanded if the name does not start with the variable, i.e. in a variable name line `'%var|any'`, `%var` is not expanded, and 2. variables in values are recognized with the pattern `[\d\w|]` and in the names with `[\d\w]`; this allows the use of so called *hierarchical attributes* in attribute values.

```
$href = { 'att1'       => '%var1',
          '%var1'      => 'string',
          'pre|%var1'  => '& "%var2" . "%var1"',
          '%var2'      => 'gnirts',
          '%pre|any'   => '100',
          'att2'       => '& %pre|any * 0.5',
        };
$atts = CTAttributes->make($href);
$atts->expand('%');
```

  changes `$href` to:

```
{ 'att1'       => 'string',
  '%var1'      => 'string',
  'pre|string' => '& "gnirts" . "string"',
  '%var2'      => 'gnirts',
  '%pre|any'   => '100',
  'att2'       => '& 100 * 0.5',
};
```

- **method** `evaluate`
  Does the same a `expand`, but evaluates variable value (and names) if they start with an ampersand (&) using `evaluate_string`.

  To continue the example above `evaluate('%')` will set `$href` to:

```
{ 'att1'       => 'string',
  '%var1'      => 'string',
  'pre|string' => 'gnirtsstring',
  '%var2'      => 'gnirts',
  '%pre|any'   => '100',
  'att2'       => 50,
};
```

- **method** `expand_string`
  Expands all variables in the given string, using the attribute hash as variable definitions. Expansion is done recursively. Expansion means a simple *search and replace*.

  The 1st parameter is the string to be expanded, the 2nd is the prefix used to mark variables (e.g. '%') and the 3rd is a pattern which variables names must match (defaults to '[\d\w]+').

  This method does not change the attribute hash itself.

```
$atts->{'%var1'} = 100;
$atts->{'%var2'} = 2;
$atts->{'%var3'} = '$var2 - 1';
$s = $atts->expand_string('%var1 + %var3', '%');
```

  sets `$s` to '100 + 2 — 1'

- **method** `eval_string`
  Same as `expand_string`. But if the string starts with an ampersand (&) the remaining string is evaluates as a perl expression. Evaluation is done recursively.

```
$atts->{'%var1'} = 100;
$atts->{'%var2'} = 2;
$atts->{'%var3'} = '$var2 - 1';
$s = $atts->eval_string('& %var1 + %var3', '%');
```

  sets `$s` to 101.

- **method** `make_tree`
  Expands all *hierarchical attributes* in the hash to be hashes of hashes ... . A *hierarchical attributes* is an attribute with bar ('|') in its name. A hash like

```
$h = { 'ini|hw.ini|Motor0|Name' => 'TL',
       'ini|hw.ini|Motor0|Type' => 'C-832' }
```

  is changed to

```
$h = { 'ini|hw.ini|Motor0|Name' => 'TL',
       'ini|hw.ini|Motor0|Type' => 'C-832',
       'ini'
         => { 'hw.ini'
                => { 'Motor0'
                       => { 'Name' => 'TL' },
                          { 'Type' => 'C_832' } } } }
```

  this allows access the the attributes

```
# as before make_tree
$val = $h->{'ini|hw.ini|Motor0|Name'};
# and
$val = $h->{'ini'}->{'hw.ini'}->{'Motor0'}->{'Name'};
# and allows iteration (that's the reason)
foreach $ini_file (keys %{$h->{'ini'}}) {
  ...
}
```

Class CT::Element

Base class for all nodes in the Classification Tree (CT).

Requirements to derived classes

must implement

- `print_tree_to_xml`

Methods and Properties

- **constructor** `new`
  Creates a new CT::Element. In dependency the the 1st parameter the returned object will be an object of the appropriate class. The 2nd parameter is the CT-object to which the element belongs, and the 3rd is the ID of the CTE-frame on which the element appears.

```
$elem = CT::Element->new("Root", $ct, $frame_id);
```

  creates an object of class `CT::Root`.

- **property** `type`
  The type of the node as string, e.g. Root", Composition", etc.

- **property** `id`
  The CTE internal ID of the node. An integer.

- **properties** `att`, `atts`, `att_inherited`, `atts_inherited`
  `att` returns the value of the attribute given by name. `atts` returns a reference to a hash containing all the nodes attributes.

  `att_inherited` does same as `att`, but if the node does not have the given attribute, all ancestors a checked for the attribute. `atts_inherited` returns a reference to a hash of all attributes available at the element itself at any ancestor. `att_inherited` and `atts_inherited` are read only properties.

  ```
  # reading an attribute named 'a1'
  $val = $el->att('a1');
  # setting an attribute named 'a1'
  $el->$el->att('a1', $val);
  ```

- **property** `frame_id`
  Returns the CTE internal ID of the frame that contains the element.

- **property** `parent_id`
  Returns the CTE internal parent ID of the direct parent element. Frame objects and elements that are roots in a frame will have a `frame_id` of -1.

- **property** `parent`
  The parent node of the element in the CT.

  `parent` skips frames, i.e. returns only real CT elements". For a element that is a root element in a frame, it returns the corresponding refinement element.

- **properties** `ancestors`, `ancestors_and_self`
  A reference to an array of all tree ancestors. The first node in the array is the root node. With `ancestors` the last is the direct parent. With `ancestors_and_self` the last is the node itself.

  The properties skip frames, see property `parent`.

- **properties** `ct_children`, `ct_children_cnt`, `ct_child`
  `ct_children` returns a reference to an array of all children of the node in the CT, except children of type 'Information' and type 'Comment'. `ct_children_cnt` returns the corresponding array size.

  `ct_child` returns the i-th child of the node. The first child has the index 0.

  `ct_children` skips frames, that means refinements does not return the corresponding frame as child, but the children of the frame. Thats why `ct_children` is overridden by class CT::Frame.

- **property** `first_ct_child`
  Returns the first child of the element which is of the given type, or `undef` if not such child is found.

  ```
  $el->first_ct_child("classification");
  ```

  returns the first classification child of `$el` or `undef` if that does not exist.

- **property** `name`
  The name of the node.

- **property** `cte_spec`
  The content of 'Specification'-property.

- **property** `cte_descr`
  The content of 'Description'-property.

Class CT::Root (a CT::Element)

The root on an CT.

Class CT::Comment (a CT::Element)

A CT comment. Currently not exported to XML.

Class CT::Information (a CT::Element)

A CT information. Currently not exported to XML.

Class CT::Composition (a CT::Element)

A CT Composition.

Class CT::Classification (a CT::Element)

A CT classification.

Class CT::ClassificationRefinement (a CT::Element)

A CT classification refinement.

Class CT::Class (a CT::Element)

A CT class.

Class CT::ClassRefinement (a CT::Element)

A CT class refinement.

Class CT::Frame

A mediating class between refinements and their children. Represents the extra view root for all children of a refinement.

Frames ID is equal to the ID of the refinement element for which the frame is the panel.

Class CT::TestObject

Base class for test objects.

- **constructor** `new`
  Creates an initializes a new `CT::TestObject` object. Takes no arguments.

- **property** `id`
  Returns or sets the CTE internal ID of the test object. An integer.

- **property** `name`
  Return or sets the name of the test object.

- **property** `cte_descr`
  Returns or sets the content of the `description` field supplied by CTE.

- **properties** `att`, `atts`
  `att` returns or sets an attribute given by its name. `atts` returns a reference to a hash containing all attribute definitions.

Class CT::TestSequence (a CT::TestObject)

A sequence of test steps.

Properties

- **constructor method** `new`
  Creates and initializes a new `CT::TestObject` object. Takes no arguments.

- **properties** `step`, `steps`
  `step` returns a `CT::TestStep` object by its position in the test sequence. `steps` returns a reference to an array of all test steps of the sequence.

```
foreach my $step (@{$seq->steps()}) {
  ... do any with the step
}
```

Class CT::TestCase (a CT::TestObject)

A test case — a collection of `CT::Class`es which constitute the test case, i.e. which has marks in the CTE combination table for the test case.

Properties and Methods

- **constructor method** `new`
  Creates and initializes a new `CT::TestCase` object.

- **property** `classes`
  A reference to an array of all `CT::Class`es which constitute the test case. There is nothing (known) promised about the order in which the classes occur.

- **property** `sub_tree`
  Returns a subtree of the classification tree with these classes an leaves only, which constitutes the test case or step.

  The tree is given as a hash reference of hash references of hash references .... Keys are the IDs of the nodes.

```
$tree = $tc->subtree;
$tree = {
         '1' => {
                   '2'  => {
                             '45' => {}
                           },
                   '7'  => {},
                   '23' => {
                             ...
                           }
                 }
        };
```

- **property** `predefined_var`
  Returns the value of a predefined variable if an appropriate name is given. The following variables are treated as predefined:

  `assert|<label>|literal`
  : The value of that variable will be the unevaluated value of the attribute `assert|<label>`. This is only useful for script generators which need the original value e.g. for error messages.

  `enabled`
  : That variable will always be predefined as 1. This can be use by script generators to skip a test step in generation. The user may signal that by setting the `enabled` attribute to 0.

- **properties** `overriding_atts1`, `overriding_atts2`
  Properties to be used by `atts_evaluated`. `overriding_atts1` shall return a reference to a hash containing attributes which override all attributes in the tree BUT NOT the attributes of the test object itself. `overriding_atts2` returns a hash reference with all attributes which override all attributes in the tree AND the attributes of the test object itself.

  This will be overridden by derived classes. The default implementation returns empty hashes.

- **property** `atts_evaluated`
  At the first call it collects all inherited, expanded and evaluated attributes of the test case, stores the result, and returns the result as a `CTAttributes` object. At the second call it returns the stored result.

  The property uses the properties `overriding_atts1` and `overriding_atts2` while collecting the attributes. Indirectly it uses the `predefined_vars` property, but predefined variables are only stores ind the result hash if they are directly referenced in any attribute definition.

- **property** `att_evaluated`
  Returns the value of an attribute given by name, using the result of `atts_evaluated` and the `predefined_vars` property.

Class CT::TestStep (a CT::TestCase)

A test step. The same as a test case, but part of a sequence of steps.

- **constructor method** `new`
  Creates and initializes a new `CT::TestStep` object. Expects the sequence to which the step belongs as argument.

- **property** `sequence`
  Returns the sequence to which the step belongs.

- **property** `step_nr`
  Returns the position of the step within the sequence. The first step of a test sequence has a `step_nr` of 1.

- **property** `predefined_var`
  Returns the value of a predefined variable if an appropriate name is given. The following variables are treated as predefined:

  `%test|step_nr`
  : The value of the `step_nr` property.

  `%test|pred_step|<variable>`
  : The `atts_evaluated` property of all preceding test steps are searched for a value for `<variable>` in backward direction. If one is found it will be the value of the named variable. Otherwise the value will be `undef`.

  `assert|<label>|literal`
  : The value of that variable will be the unevaluated value of the attribute `assert|<label>`. This is only useful for script generators which need the original value e.g. for error messages.

  `enabled`
  : That variable will always be predefined as 1. This can be use by script generators to skip a test step in generation. The user may signal that by setting the `enabled` attribute to 0.

- **properties** `overriding_atts1`, `overriding_atts2`
  `overriding_atts1` returns the attributes of the test sequence to which

the test step belongs. They have to override the attributes in the classes constituting the test step, but not the attributes at the test step itself.

`overriding_atts2` returns an empty hash.

C.2 Perl Module ScriptGenerator.pm

Class ScriptGenerators::ScriptGenerator

This module implements only one class. This class may be used as a base class for implementations of script generators used by the `cte2cpp` test script generation tool.

The only method which MUST at least be defined by derived classes is the `generate`-method which takes a single `CT::TestCase` or a `CT::TestSequence` object as argument.

This class is intended to collect reusable methods for test script generation.

Methods and Properties

- **contructor method** `new`
  Takes no arguments. If your derived class implements its own `new`, this should this method like this:

```
sub new {
  my $this = shift;
  my $class = ref($this) || $this;
  my $self  = $class->SUPER::new;

  ... do any new things

  return bless ($self, $class);
}
```

- **property** `output`
  This returns or sets a output filehandle as default target for generating procedures. This property is set to `*STDOUT` by default.

```
open(FILE, qq(>$fn)) || die qq(can't open $fn ($!));
$gen->output(*FILE);
```

- **property** `testobj`
  Returns or sets the test object for which the script shall be generated.

- **property** `generate`
  This method is not defined by this class and has to be implemented by derived classes. This method should implement the job of test script generation for the given test object.

```
sub generate
{
  my ($self, $testobj) = @_;

  $self->testobj($testobj) if (defined $testobj);

        ... to be redefined
}
```

- **method** `generate_ini_file`
  This method requires as argument a `CTAttributes` object as returned by the `atts_evaluated`-property of the test object. If , e.g., the attribute definitions at the test object in the CTE-file contains the following attributes:

```
ini|hw.ini|Motor0|Name: Tilt
ini|hw.ini|Motor0|Type: C-832
ini|hw.ini|Motor1|Name: CC
ini|hw.ini|Motor1|Type: C-832
ini|dp.ini|Sp|User: ab
ini|dp.ini|Sp|Type: NULL
```

  the `CTAttributes` will contain a part like that:

```
{ 'ini' => {
              "hw.ini" => {
                            "Motor0" => {
                                          "Name" => "Tilt",
                                          "Type  => "C-832"
                                        }
                            "Motor1" => {
                                          "Name" => "CC",
                                          "Type  => "C-832"
                                        }
                          },
              "dp.ini" => {
                            "Sp" => {
                                      "User" => "ab",
                                      "Type" => "NULL"
                                     }
                          }
            }
}
```

  and `generate_ini_file` will generate the following ini-files:

```
;file hw.ini
[Motor0]
Name=Tilt
Type=C-832
[Motor1]
Name=CC
Type=C-832

;file dp.ini
[Sp]
User=ab
```

  Note the special handling of attributes with a value of `NULL`. They are not written out.

- **method** `generate_ini_structure`
  This takes the same information as `generate_ini_file`. It does not generates files but an output which can be used for a data driven generator for the needed files. Continuing the example above

```
$generator->generate_ini_structure('ini_data',
                              $to->atts_evaluated(), *OUTPUT);
```

  will write the following to the `OUTPUT` filehandle:

```
INI_BEGIN(ini_data)
INI_ENTRY("hw.ini", "Motor0", "Name", "Tilt")
INI_ENTRY("hw.ini", "Motor0", "Type", "C-832")
INI_ENTRY("hw.ini", "Motor1", "Name", "CC")
INI_ENTRY("hw.ini", "Motor1", "Type", "C-832")
INI_ENTRY("dp.ini", "Sp", "User", "ab")
INI_ENTRY("dp.ini", "Sp", "Type", NULL)
INI_END
```

  This can be used as a `C/C++` macro. With an appropriate macro definition this may be used to create a structure that can be used by a test driver to generate the needed files while test execution. For example:

```
typedef struct {
  char* filename;
  char* section;
  char* name;
  char* value;
} TIniEntry;
#define INI_BEGIN(name) \
   TIniEntry name[] = {
#define INI_ENTRY(filename, section, name, value) \
   { filename, section, name, value },
#define INI_END  \
   { 0, 0, 0, 0} };
```

  or:

```
#define INI_BEGIN(name)
#define INI_ENTRY(file, section, name, value) \
   WritePrivateProfileString(section, name, value, file);
#define INI_END
```

  Note again the special handling of attributes with a value `NULL`. The 3rd argument defaults to the `output` property.

- **method** `generate_teststeps_code`
  This method expects as first argument a reference to an array of test objects (`CT::TestStep` or `CT::TestCase`) and as second an output filehandle. The second argument defaults to `output` property.

This is intended to generate code in the test script, e.g. to proceed test steps. This uses the so called `step` attributes at the test objects given in the first argument.

If a first test object has the following evaluated attributes

```
step|1: LOG_ANY_IMPORTANT("message")
step|2: DO_THE_TEST_STEP("anArgument")
```

and a second has something like that

```
step|a: cout << "message";
step|b: call_a_function(100, "abc");
```

`generate_teststeps_code` will produce the following on the output:

```
LOG_ANY_IMPORTANT("message")
DO_THE_TEST_STEP("anArgument")

cout << "message";
call_a_function(100, "abc");
```

The second part of the attribute name (the part after '|') is used to sort the `step` attributes and the value of the attribute is written to a single line of the output, indented with a tab character.

The method uses an attribute named `enabled`. If the attribute is set to 0 the test object is skipped. This is intended to give the user the possibility to avoid the output of a test step in a sequence without deleting them or changing attributes.

- **methods** `evaluate_assertions_for_teststep` **and** `*_for_testseq`
  These methods are expecting both a test object (a `CT::TestStep` or a `CT::TestSequence` respectively as argument. They use the expanded `assert` attributes to evaluate there values as (boolean) Perl expressions. If the evaluation returns `0`, `'0'`, `''` (empty string) or `undef` the assertion is treated as failed. Than a warning is printed to `STDERR`. E.g., the following attributes at a test case

  ```
  assert|position : -1000 <= 100 && 100 <= 1000
  assert|direction: -1 == 1
  ```

  will produce a warning like

  ```
  warn: assertion direction failed
        '%direction == %upwards' (-1 == 1)
  ```

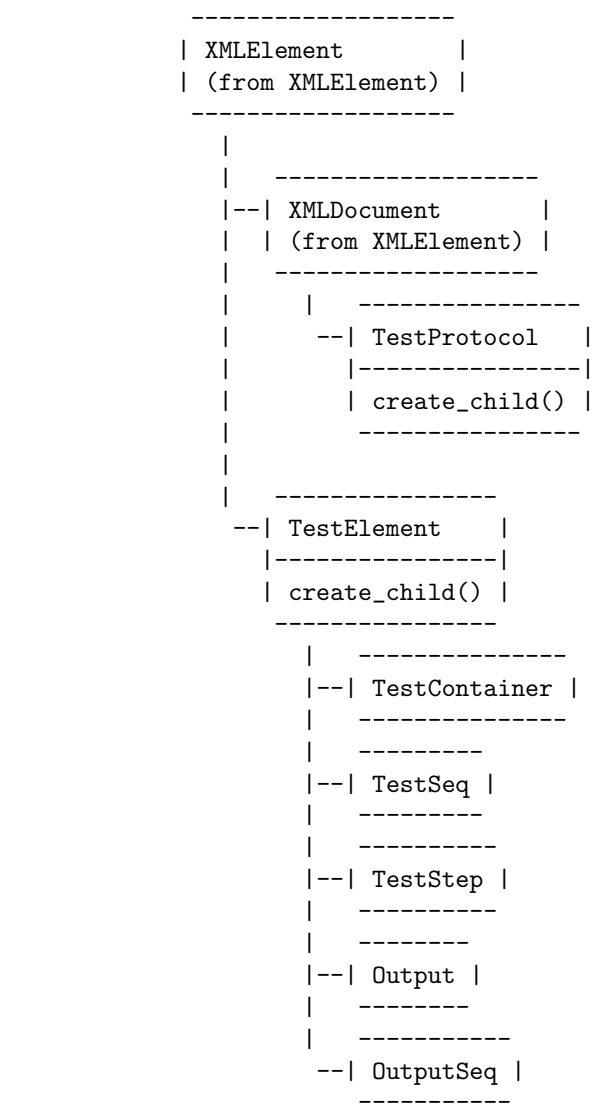
  assuming that the unexpanded attribute definition of `assert|direction` is `%direction = %upwards` and `%direction` is set to -1 and `%upwards` to 1. The second part of the attribute name is thought as an identifier string.

C.3 Perl Module TestProtocol.pm

Overview

This module uses the Perl module `XMLElement`. It derives classes from the the class `XMLElement` specializing them for the needs to evaluate test results. It provides the import of log and baseline files and implements the algorithm to extract the output data for comparing them while test evaluation.

The classes of the module are the following:



All classes are derived from `XMLElement` and have to implement some common methods and properties:

- **method** `create_child`
  This method determines how XML child elements of an element are crea-

ted while parsing. Actually it maps element names to classes. This is implemented at the classes `TestElement` and `TestProtocol` only.

- **property** `accepts_data`
  this property has to be 1 if the XML element can contain data (i.e. PCDATA) and 0 if not.

- **properties** `static_qname`, `qname`
  `static_qname` determines whether the class redefines the `qname` property. The `qname` property returns the qualified name of the XML element.

In the current state of the module the following elements of a test protocol or a baseline file are imported: `test-log`, `test-bas`, `test-seq`, `test-step`, `output`, `regexp`, `output-seq`, `error`. For some elements exist specialized classes, e.g. `Output` and `OutputSeq` for the elements `output` and `output-seq`. Some other elements are mapped to the generic classes `TestElement` and `TestContainer`, e.g. `test-log` and `regexp`.

The expected structure of the XML data describe the following document type definition. Any other XML elements in the XML data are ignored.

```
<!-- actual output -->
<!ELEMENT test-log (title, date, test-step*)>
<!ATTLIST test-log id CDATA #REQUIRED>

<!-- target output -->
<!ELEMENT test-bas (title, date, test-step*)>
<!ATTLIST test-bas id CDATA #REQUIRED>

<!ELEMENT test-step (title,
                     (output|output-seq|error)*)>
<!ATTLIST test-step number CDATA #IMPLIED>

<!ELEMENT output (#PCDATA)>
<!ATTLIST output context (all|baseline|targetline) 'all'>
<!ATTLIST output comp (regexp) #IMPLIED>

<!ELEMENT output-seq (output|output-seq|error)*>
<!ATTLIST output-seq min-occur CDATA #REQUIRED>
<!ATTLIST output-seq max-occur CDATA #REQUIRED>

<!ELEMENT title   (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT error   (#PCDATA)>
```

Class `TestElement` (a `XMLElement`)

Base class for all test protocol element classes. Redefines the `create_child` method.

May be used for all element types which contain data as a generic implementation and need no specialized behavior.

Class `TestProtocol` (a `XMLDocument`)

The class for the protocol itself. This is the container for all logging and baseline information.

```
my $protocol = new TestProtocol;
$protocol->create_from_file("$any_path/$test_id.LOG");

# to do any thing with every test-step
my $iter = XMLElementIterator->new($protocol, 'test-step');

while (my $step = $iter->next() ) {
  ....
}
```

Class `TestContainer` (a `TestElement`)

May be used as a generic implementation for all elements which does not contain data.

Class `TestSeq` (a `TestElement`)

Container for a number of test steps.

Class `TestStep` (a `TestElement`)

Container for all the output of a test step. In the current test system a test step is the level on that test evaluation happens. That means that the collected output of a test step is evaluated as whole.

All the implementations of collecting the output delegate all the work to the containing elements and collects their return values.

Methods and Properties

- **property** `title`
  Returns the content of the title element of the test step.

- **property** `number`
  Returns the content of the number attribute of the test step.

- **property** `target_output_pattern`
  Collects all output of the test step and returns it as a regular expression. It can be used by a comparator to compare the `actual_output` (see below) against this expression.

  It takes as its single argument the name of a so called *context*. This is used to filter the output by its `context` attribute. That means that in the regular expression all output will occur which has no `context` attribute or a context attribute equal to the given argument.

- **property** `actual_output`
  Returns a string containing all output of the test step. This can be used

to evaluate the output by matching this string with the regular expression returned by `target_output_pattern`.

It accepts the same `context` argument as `target_output_pattern`.

- **property** `output_diffable`
  Returns a string representing the output of the test step for comparing the string with an other one using a tool like *diff*.

  It expects the same `context` argument as `target_output_pattern`.

Class `Output` (a `TestElement`)

A unit out output.

Properties and Methods

- **property** `context`
  The value of the `context` attribute or `'all'` if the element has no `context` attribute.

- **property** `actual_output`
  See `actual_output` at class `TestStep`. Returns the data content of its own and its children surrounded by `o` tags if its `context` attribute is `all` or equal to the argument. Otherwise it returns an empty string.

- **property** `target_output_pattern`
  See `target_output_pattern` at class `TestStep`. If its `context` attribute is `all` or equal to the given argument it transforms the contained data to an regular expression surrounded by `o` tags. If the element has a `comp` attribute with `regexp` the data is not transformed and embedded `regexp` elements are handled the same way.

  ```
  <output>12[0-9]</output>
  ```

  is transformed to

  ```
  <o>12\[0-9\]</o>
  ```

  and matches exactly the string `'12[0-9]'`. But

  ```
  <output comp="regexp">12[0-9]</output>
  ```

  is transformed to

  ```
  <o>12[0-9]</o>
  ```

  and matches all string representing the numbers 120 to 129.

- **property** `output_diffable`
  See `output_diffable` at class `TestStep`. If the `context` attribute is appropriate the content is surrounded by `output` tags and returned as a single line.

Class `OutputSeq` (a `TestElement`)

An output sequence, i.e. a container for `output` elements which occur in the logged data optionally or repeated a changing number of times. Output sequences occur in *baseline* data only. That's why this class does not implement the `actual_output` method.

Properties

- **property** `min_occur`
  The minimal number of occurrences the sequence is expected in the logged data. Returns the value of the `min_occur` attribute or `0` of it is implied.

- **property** `max_occur`
  The maximal number of occurrences the sequence is expected in the logged data. Returns the value of the `max_occur` attribute or `*` of it is implied.

- **property** `target_output_pattern`
  See `target_output_pattern` at class `TestStep`. If its `context` attribute is `all` or equal to the given argument it collects the patterns of all containing `output` elements, surrounded with parenthesis and with a appended quantifier according to Perl regexp syntax.

  ```
  <output-seq min-occur="2" max-occur="24">
   <output>a</output>
   <output>b</output>
  </output-seq>
  ```

  is transformed to

  ```
  (<o>a</o><o>b</o>){2,24}
  ```

- **property** `output_diffable`
  See `output_diffable` at class `TestStep`.

C.4 Perl Module Comparator.pm

Overview

This module provides an algorithm to compare the *actual output* of test execution with the *target output*. Both have to be given as XML data. It uses the objects provided by the module `TestProtocol` to access the information in the given files. See there for the expected structure of the data.

The module does not implement a class but a simple public method named `compare`. This method expects the following arguments:

1. build ID
2. root directory of the test system
3. test ID
4. package label
5. target data label
6. name of the file containing the *actual output*
7. name of the file containing the *target output*
8. name of the file where the results of comparison are to be stored

First it reads two files: the *actual output* and the *target output* and than it does the comparison `test-step` by `test-step`. For each `test-step` it extracts the output data in the *target output*, transforms it to an regular expression, extracts the output data in the *actual output*, transforms it to a simple string and tests whether the string matches the regular expression. If it does, the `test-step` is treated as successful. If not, an *error description file* is created for that `test-step` (s. `save_diff`).

If the *target output* contains different output for the *baseline* — the output produced while the last test — and the *targetline* — the output that should be produced — the test described above is done for both. Differences between *baseline* and *targetline* are marked with the `context` attribute (s. module TestProtocol).

It returns the number of `test-step`s with an error.

Private Methods

- **method** `count_logged_errors`
  Counts the `error` elements in the XML tree given by a single XML element and returns the number.

- **method** `save_diff`
  Creates an *error description file*. It expects five arguments. The 1st is the test ID. The next two are the test objects (`CT::TextStep` or `CT::TestCase`) to be compared, i.e. the comparator decided that their output doesn't match. The 3dr argument is the `context` attribute to filter the output and the last is the name of the error description file to be created.

The method first extracts the *diff* able output of both test objects stores it in two files and uses `diff` to find the differences. The output of `diff` is transformed into a simple XML structure that may be used to visualize the differences. That XML is stored in the given *error description file*.

The `diff` tool must be reachable via the `PATH` environment variable.

- **method** `diff_nice`
  Interprets the left and right input and the output of the `diff` tool and returns XML to represent the differences.

```
$bas =
"<o>value1</o>
 <o>value2</o>
 <o>value3</o>
 <o>
 line1
 line2
 line3
 </o>";

$log =
"<o>value1</o>
 <o>value3</o>
 <o>value4</o>
 <o>
 line1
 line3
 line4
 </o>";

$dif =
"2d1
 < <o>value2</o>
 3a3
 > <o>value4</o>
 6d5
 < line2
 7a7
 > line4";
```

`diff_nice($bas, $log, $dif)` returns

```
"<o>value1</o>
 <o sty='deleted'>value2</o>
 <o>value3</o>
 <o sty='added'>value4</o>";
 <o>
 line1
 <deleted>line2</deleted>
 line3
```

```
<added>line4</added>
</o>";
```

- **method** `output_deleted`
  Takes a string (a line of diff'ed output) and marks it as deleted. If the line contains a starttag the attribute `sty='deleted'` is added. If it is a normal line it is surrounded with `deleted` tags and if it is an endtag nothing is done, i.e.

```
output_deleted("<output>")
returns
"<output deleted>"

output_deleted("output")
returns
"<deleted>output</deleted>"

and output_deleted("</output>")> returns
"</output>"
```

- **method** `output_added`
  Does the same as `output_deleted` but uses `added` as mark.

C.5 Perl Script run_test.pl

Description

This executes a test and compares the *actual output* with the *target output*. It has the following options:

`--root-dir=<directory>`
: Specifies the root directory of the test system. This is the directory where a directory for the test package and the build can be found.

`--test-pkg=<package>`
: Specifies the name of the test package. This is used to find the directory of the test package within the root directory.

`--build=<build>`
: Specifies the name of the build to be tested. This is used to find the directory with the binaries of the application to be tested and the test programms within the root directory.

`--test=<test-id>`
: Gives the ID of the test to be executed.

`--test-list=<filename>`
: Specifies a file name containing a list of IDs of tests to be executed. The file has to contain a single line for each test ID.

`--baseline=<baseline>`
: Gives the name of the target output version. This is used as a directory name which is expected in the package directory.

`--execute-only`
: Suppresses the comparison.

`--compare-only`
: Suppresses the test execution.

One of the options `--test` or `--test-list` is required.
With the parameters the following directories and files are expected:

`<directory>/<package>`
: The test package directory.

`<directory>/<package>/<baseline>`
: The directory containing the version of *target output* which is used in the comparison.

`<directory>/<package>/<baseline>/<test-id>.BAS`
: The file containing the *target output* of the test `test-id`

`<directory>/<build>`
: The directory containing the build of the application under test and the test programms.

`<directory>/<build>/<test-id>.EXE`
: The test programm to be executed.

`<directory>/<build>/<test-id>.LOG`
: The *actual output* of the execution of the test programm. If `--compare-only` is given, this file must always exist.

C.6 Perl Script cte2cpp.pl

Description

This programm takes a CTE file, imports it, looks for a script generator, creates it and let them generate the the needed things for a specified test case or sequence.

It has two mandatory and an optional parameters:

`--cte-file=<file>`
: Specifies the CTE file from which the test scripts has to be generated.

`--test-id=<id>`
: Gives the identifier of the test case or the test sequence for which the script should be generated. To identify a test case the `test_id` attribute is used.

`--generator=<name>`
: Gives the name of the script generator to be used.

If no script generator is given with the `--generator`-option, this programm looks for a `ScriptGenerator` attribute as the *root* node of the classification tree to determine the script generator to be used. The value of that attribute or the option is interpreted as a module name. So, if the value is `any` all include directories are searched for a perl module named `any.pm`. An appropriate directory can be given via the `/I` option of the perl interpreter. For example:

```
perl -I ../bin/lib/ScriptGenerators ../bin/cte2cpp.pl
                --cte-file=m_rpl.cte --test-id=RPL001
```

adds `../bin/lib/ScriptGenerators` to the include directories. Uses the given `m_rpl.cte` file in the current directory, searches it for a test case or sequence with a `test_id` attribute with a value of `RPL001` and generates with the help of a hopefully found script generator the test script *file*(s).

This tool requires the Perl module CT (Classification Tree).

C.7 Perl Script cte2xml.pl

Description

This tool converts a CTE file to an XML document using the services of the `CT` Perl module. It has three option:

`--cte-file=<file1>`
: Specifies the CTE file to be converted. This option is mandatory.

`--xml-file=<file2>`
: Specifies the XML file to be exportet. This option is optional. It defaults to `<file1>.xml`

`--extent=<number>`
: This option controls the extent of the exported XML. Currently if `number` is greater than `0` for each test step of a sequence all evaluated attributes are exportet to. This produces bigger XML files and takes a little more time but may sometimes usefull finding bugs in the test specifications. This option defaults to `0`.

C.8 Perl Script cte2latex.pl

Description

This tool reads a CTE file containing a classification tree and transforms it into a LaTeX file for printing a textual form of the tree for documenting purposes. Reading and evaluating the CTE file is done using the `CT` module.

It has three option:

`--cte-file=<file1>`
: Specifies the CTE file to be converted. This option is mandatory.

`--tex-file=<file2>`
: Specifies the file to be exportet. This option is optional. It defaults to `<file1>.tex`

`--tree-only`
: This flag suppresses the export of attributes and descriptions for the tree nodes.

LaTeX-Output

The generated LaTeX output looks like the following example:

```
\begin{ct}{<name of the root node>}
  \begin{ct-attributes}
    \ctAttribute{<attribute1>}{<value1>}
    \ctAttribute{<attribute2>}{<value2>}
  \end{ct-attributes}
  \begin{ct-classification}{<name of classification>}
    \begin{ct-class}{<name of class>}
      \ctDescription{<content of description>}
      \begin{ct-classification-refinement}{<name of refinement>}
        \ctDescription{<content of description>}
        \begin{ct-attributes}
          \ctAttribute{<attribute1>}{<value1>}
          \ctAttribute{<attribute2>}{<value2>}
        \end{ct-attributes}
      ...
      \end{ct-classification-refinement}
    \end{class}
  ...
  \end{ct-classification}
...
\end{ct}
```

Refinement nodes in the classification tree with only one child node are suppressed in the output. These nodes are only placeholder in the graphical representation and are treated as obsolete in an textual representation.