|  |
| --- |
|  |
| **Title\*:** | Ericsson input to TTCN-3 OO planning |
|  |  |
| from **Source**\*: | Ericsson LM |
| Contact: | György Réthy  |
|  |  |
| input for **Committee**\***:** | MTS |
|  |  |
| Contribution **For\*:** | Decision |  |  |
|  | Discussion | **X** |  |
|  | Information |  |  |
|  |  |
| Submission date**\***: | 2017-01-19 |
|  |  |
| Meeting & Allocation: | **MTS#70** -  |
| Relevant WI(s), or deliverable(s): |   |
|  |

**ABSTRACT:***One of the tasks of new STF TTCN-3 Evolution 2017 is to develop the object oriented extension of the language. TC MTS has to set clear requirements and guidance for the STF, and has to agree on the steering of this task. This contribution collects the first set of inputs from Ericsson.*

# Purpose of the OO extension

Give a new momentum to the language. Today test suite developers are facing several problems that are difficult to solve in TTCN-3 as today, thus causing hard-to-maintain and slow-to-develop code. This is typical for solutions, where test cases are written on top of big, in most cases multilayer frameworks. In Ericsson this is the case for both functional testing (the biggest framework+test cases consist more than 3 Mline of code) and test applications developed for performance testing.

The language contains “traces” of OO features, however no conceptual frame is defined for these features and hence there is no easy-to-understand way of using them.

# Problems identified up to date

* It is difficult to develop **functionality** that need to work with **different types of data** (e.g. hashmaps, FSM descriptions etc.)
* No **logical packaging** of data and functions manipulating them is possible (e.g. define a SIP user as a logical entity with all its data and available functionality). This results more complex code and slower development than the task would require in other languages.
* **Information hiding** is not well solved, TTCN-3 visibility today allows difficult and incomplete solutions only.
* Not possible to **automatically initialize** data and **components** (e.g. to initialize a component simulating a SIP user with the data of a concrete user or pool of users).
* **Frameworks are difficult to customize**: today callback functions are used for this purpose, however they result hard-to-read code and slow down test case development, e.g. if a new callback insertion points is required, the framework developers have to be asked to add it, that slows down .
* Collecting existing problems is still ongoing…

# Proposed approach

To be discussed and agreed in TC MTS.

Our high-level proposal is: collect the OO concepts and features from existing OO languages and analyse their usability in the context of TTCN-3 (benefits-dangers/drawbacks). Add only concepts and language elements that solves a real problem, has clear benefit and doesn’t break the requirements set for the extension by TC MTS. Does not include an OO feature in the language just because other OO language(s) support it, or for its technical beauty! Keep it as simple and clear as possible!

It shall be clearly understood and defined in the standard, what will be the “differentiating factor” to other OO languages (see e.g. initial input to requirements in the next clause).

# Requirements, “forbidden fruits”

The basic concepts and usability requirements of TTCN-3 (of a test language) still need to be followed. Therefore, OO concepts/features being analysed shall be evaluated according to a set of well-defined criteria and introduce only which fulfils the criteria (or introduce it with changes to fulfil the criteria).

Below a few items are listed as the initial input to this part of the discussion.

* It shall be consistent with existing language features, maximum backward compatibility.
This does not mean no change at all, some features/syntactical solutions may become deprecated, if unification of the language concepts requires this, but e.g. runtime behaviour of existing language elements shall not be changed.
Backward compatibility is crucial for the introduction of the new features into the existing code base with minimal effort, which is, on the other hand, is the key to introduce OO at all. However, unifying the concepts for the different objects may result in syntactical changes. These cases should be thoroughly discussed and agreed in TC MTS, supporting the existing syntax should be guaranteed for a certain amount of time and the refactoring of existing code should be easy-to-automate.
* It is very important that users can understand the runtime behaviour by reading the source code. One of the advantages of TTCN-3 today – beyond being a standard and its testing specific nature – is that it’s runtime behaviour can be understood by users with minimal learning curve and without running and debugging the test suite.
E.g. dynamic typing, implicit overriding & inheritance etc. should be “forbidden fruits”. Keep everything explicit as much as possible.
* Maximum compile time check-ability shall remain one of the most important advantages of TTCN-3.
It is often the case that testers shall develop test cases without access to the IUT/SUT and when they get access to it, the machine time available is limited; this time shall be used to test/debug the IUT/SUT and to debug the test code! Also when thinking about e.g. TDD (test driven development), tests are written before the feature itself is developed, hence the test code shall be “as error-free as possible”.
* Help the user to avoid unintended errors when writing the code. This is related to both compile-time check-ability and language features.
For example, we have avoided overloading in the current language, because different functions with different number of parameters can easily lead to a situation, where already existing function calls suddenly start calling a different function with a different behaviour, or simply it is difficult to choose for the (framework) user which function to be called.
Also, operators, built-in operations and pre-defined functions should not be overridden.
* Avoid pointers beyond the existing active object “handle”s (variables of component and default types). Keep “assignment” as the only concept of value/template passing at the TTCN-3 level; tools can, of course, optimise the execution of assignments in different contexts.
* It should be possible to close the derivation path (see e.g. clause 1.2.3), i.e. allow framework developers to close their code, where they consider this to be necessary.

# Possible object classes

It is not the purpose of this contribution to go into a detailed proposal at this stage, rather to start collecting ideas and positions, to inspire the discussion.

## Data classes

This could be the major addition to the language.

Classes could be a new, additional kind in the TTCN-3 data-type system, which can include members: constants, variables (of exiting TTCN-3 or ASN.1 types), functions, an “initializator” (constructor) and a “finalizer” (destructor). Restrictions of member data and in member functions regarding the use of runs on/system clauses, components, ports and timers should carefully be considered and specified in the standard.

In developing the concept and functionality of classes, the approach agreed in TC MTS shall be followed.

Pros

* Unified concepts in the language
* Opportunity to solve the current language issues in clause 2 Problems identified up to date (and collected later)

Cons

* Overcomplicating the solution may have reverse effect, instead of speeding up TTCN-3 code development may lead to unpredicted runtime problems and at the end slow down the testing activities.

Other

* Record/set type extensibility
This feature would not really be part of the OO features, but similar to those and would add a lot to the language’s usability. Record/set types could be extended (modified?) with new fields: syntax-wise could be like template modification or component extensions

(as the initial suggestion, classes should be “closed” entities, i.e. scope units, members functions of which should see and manipulate other class members only (including all inheritance, accessibility etc, functionality, of course), thus no other objects or object instances like components, ports and timers should be allowed).

## Component, port and timer classes

Built-in classes of these objects could be defined by the standard, including the existing component, port and timer operations as functions of the appropriate built-in class. User component, port and timer definitions could be

Pros

* Unified concepts in the language
* Simplified description of these objects
* If a unified approach is used for all these objects, lots of TTCN-3 keywords could be released as keywords, because they could become functions of the base objects (start, stop, done, kill, killed, running, alive, ….send, receive, call, getcall, reply… timeout, read etc.)

Cons

* Only very few of the above operations uses function-like syntax (e.g. starting a timer with a one-time timer value), the other operations don’t use the “()”; this problem could be overcome e.g. be deprecating the old operation definitions (old syntax), but tools will support them further on at least for a certain amount of time. Introduce the new function-like syntax for these base classes. Tools could also support automatic re-factoring, making the switching easier for the user.
* “Implicit derivation” could be an unusual construct that would require a good conceptual background.

### Component classes

Pros

* Provides basis for extensibility already that could be re-used for other objects classes.
* Would allow automatic initialization and “finalization” of component instances. Today the unavailability of these features causes lots of problems for the developers.
* Would allow defining component variables and functions manipulating them in a packed way.
* Could allow solving the component definition visibility issue ([CR6801](http://forge.etsi.org/mantis/view.php?id=6801)) in a consistent way.

Cons

* None identified, but existing component extensibility shall be “embedded” into the new concept in a consistent way.

### Port classes

Pros

* Translation functions for translational ports could be defined within the port itself, i.e. a more “packed” and control/able way for the user.
* Could allow port extensibility, where additional framework layers or the test cases want to add additional types to the port message/procedure lists (e.g. for control ports).

Cons

* None identified, but existing port parameterization shall be “embedded” into the new concept in a consistent way.

### Timer classes

Each timer declaration could be an instance of the built-in timer class that should be a final, not inheritable class (in principle, the user could define further timer operations if inheriting from the built-in timer class is allowed, which could be useful, however raises the implement-ability question).

Pros

* No additional to those in clause 1.2 identified.

Cons

* No additional to those in clause 1.2 identified.