Final draft ETSI ES 203 790 V1.1.1 (2018-10)

Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Object-Oriented Features

Reference

DES/MTS-203790-00F_ed111

Keywords

language, TTCN-3

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

The present document can be downloaded from:
http://www.etsi.org/standards-search

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the only prevailing document is the print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx

If you find errors in the present document, please send your comment to one of the following services:
https://portal.etsi.org/People/CommiteeSupportStaff.aspx

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This final draft ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS), and is now submitted for the ETSI standards Membership Approval Procedure.

The present document relates to the multi-part standard ETSI ES 201 873 covering the Testing and Test Control Notation version 3, as identified in ETSI ES 201 873-1 [1].

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document defines the support for object-oriented features in TTCN-3. TTCN-3 can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of OMG CORBA based platforms, APIs, etc. TTCN-3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. The specification of test suites for physical layer protocols is outside the scope of the present document.

TTCN-3 packages are intended to define additional TTCN-3 concepts, which are not mandatory as concepts in the TTCN-3 core language, but which are optional as part of a package which is suited for dedicated applications and/or usages of TTCN-3.

While the design of TTCN-3 package has taken into account the consistency of a combined usage of the core language with a number of packages, the concrete usages of and guidelines for this package in combination with other packages is outside the scope of the present document.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references,only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference/.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ETSI ES 201 873-4: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".

[3] ETSI ES 201 873-5: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

[4] ETSI ES 201 873-6: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references,only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[i.2] ETSI ES 201 873-8: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.3] ETSI ES 201 873-9: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.4] ETSI ES 201 873-10: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification".

3 Definition of terms and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI ES 201 873-1 [1], ETSI ES 201 873-4 [2], ETSI ES 201 873-5 [3] and ETSI ES 201 873-6 [4] apply.

3.2 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI ES 201 873-1 [1], ETSI ES 201 873-4 [2], ETSI ES 201 873-5 [3] and ETSI ES 201 873-6 [4] apply.

4 Package conformance and compatibility

The package presented in the present document is identified by the package tag:

`"TTCN-3:2018 Object-Oriented features"` - to be used with modules complying with the present document.

For an implementation claiming to conform to this package version, all features specified in the present document shall be implemented consistently with the requirements given in the present document and in ETSI ES 201 873-1 [1] and ETSI ES 201 873-4 [2].

The package presented in the present document is compatible to:

- ETSI ES 201 873-1 [1], version 4.10.1;
- ETSI ES 201 873-4 [2], version 4.6.1;
- ETSI ES 201 873-5 [3], version 4.8.1;
- ETSI ES 201 873-6 [4], version 4.9.1;
- ETSI ES 201 873-7 [i.1];
- ETSI ES 201 873-8 [i.2];
- ETSI ES 201 873-9 [i.3];
- ETSI ES 201 873-10 [i.4].

If later versions of those parts are available and should be used instead, the compatibility to the package presented in the present document has to be checked individually.

5 Package Concepts for the Core Language

5.0 General

This package defines objec-oriented features for TTCN-3, i.e. it extends the TTCN-3 core language (ETSI ES 201 873-1 [1]) with well-known concepts from object-oriented programming and modelling languages. This package realizes the following concepts:

- classes (i.e. class definition, scope rules, abstract and external classes, refinement, constructors, destructors, methods, visibility, and built-in classes),
- objects (i.e. ownership, object references, select class-statement, dynamic class discrimitation, and casting), and
- exception handling (i.e. ability to define exception handling for functions, external functions, altsteps, and test cases).

5.1 Classes and Objects

5.1.0 General

This clause introduces the concepts of class types and their values, called objects as well as the operations allowed to be applied to these objects.

5.1.1 Classes

5.1.1.0 General

Syntactical Structure

```
[public | private]
type [external] class [@final |@abstract]
Identifier [extends Identifier]
[runsOnSpec] [systemSpec] [mtcSpec]
"{" {ClassMember} "}"
[finally StatementBlock]
```


Semantic Description

A class is a type where the values are called objects. A class can declare fields (variables, constants, templates, ports, timers) and methods as its members. Each member name inside the class shall be unique, there is no overloading. The private and protected fields and methods are only accessible by the methods of the class, while the public members of the class can be accessed also from behaviour not defined in the class. The private members of the class can be accessed directly only by members of the class itself. All members which are neither private nor public are protected and can also be accessed by members of subclasses.

A class can extend another class. The extended class is called the superclass, while the extending class is called the subclass. The resulting type of a class definition is the set of object instances of the class itself and all instances of its direct or indirect subclasses. A subclass is a subtype of its direct and indirect superclasses and its object instances are type compatible with them. If a class does not explicitly extend another class type, it implicitly extends the root class type object. Thus, all classes are directly or indirectly extensions of the object class.

A class can have optional runs on, mtc and system clauses. This restricts the type of component context that can create objects of that class and all methods of this class. If a class does not have one of these clauses, it inherits it from its superclass, if the superclass has one. If the superclass has or inherits a runs on, mtc or system clause, the subclass may declare each of these clauses with a more specific component type than the one inherited. The function members of classes shall not have runs on, system or mtc classes but inherit them from their surrounding class or its superclasses.

Restrictions

a) Templates are not allowed for class types.

b) Passing of object references to the create operation of a component type or a function started on another component is not allowed.

c) No subtyping definition is allowed for class types via the normal subtype definition.

d) No local/global constants or module parameters of class type or containing class type fields or elements are allowed.

e) Class type can not be the contained value of an anytype value.

f) The functions of a class shall not have a runs on, mtc or system clause.

g) The runs on type of a class shall be runs on compatible with the runs on type of the behaviour creating a class.

h) The runs on type of a class shall be runs on compatible with the runs on type of the superclass.

i) The mtc and system type of a class shall be mtc and system compatible with the mtc and system types of the superclass, respectively.

5.1.1.1 Scope rules

Class constitutes a scope unit. For the uniqueness of identifiers, the rules specified in the clause 5.2.2 of ETSI ES 201 873-1 [1] apply with the following exceptions:

a) Identifiers from the higher scope can be reused for member declarations. A reference to a reused identifier without a prefix occurring inside a class scope shall be resolved as a reference to the class member. In order to refer to the declaration on the higher scope, the identifier shall be preceded with a module name and a dot (".").

b) Identifiers of member declarations can be reused inside methods for formal parameter and local declarations. A reference to a reused identifier without a prefix occurring inside a class method shall be resolved as a reference to the formal parameter or local declaration. In order to refer to the member declaration, the identifier shall be preceded with the `this` keyword and a dot.

c) Reusing identifiers of members of the component type specified in the runs on clause of the class for members and inside methods for formal parameters and local declarations is not allowed.

EXAMPLE:

```
module ClassModule {
    const integer a := 1;

    type class MyClass() {
        const integer a := 2;
        function doSomething (integer a := 3) {
            log(a); // logs 3 (for the default value)
            log(this.a); // logs 2
            log(ClassModule.a); // logs 1
        }
        function doSomethingElse () {
            log(a); // logs 2
            log(this.a); // also logs 2
            log(ClassModule.a); // logs 1
        }
    }

}
```

5.1.1.1 Abstract classes

A class can be declared as @abstract. In that case, it is allowed that it also declares abstract member functions who shall be defined by all non-abstract subclasses. An abstract method function has no function body but can be called in all concrete instances of subclasses of the abstract class declaring it. Other members of the abstract class or its subclasses may use the abstract functions as if it was concrete where at runtime the concrete overriding definition will be used.

NOTE 1: Abstract classes are only useful as superclasses of concrete classes.

Restrictions

a) Abstract classes cannot be explicitly instantiated.

b) If a class that is not declared abstract extends an abstract class, all methods that have no implementation in the superclass shall be implemented in this class.

NOTE 2: Variables of an abstract class type can only contain references to instances of non-abstract subclasses.

5.1.1.2 External classes

A class may also be declared as external. In that case, all members shall be external functions without a function body. It is allowed to omit the external keyword from these function declarations. When instantiating an external class, the object being created is provided by the platform adapter and the method calls to the object are delegated via the platform adapter to the corresponding method of the external object.

NOTE 1: External classes are a way to use object-oriented library functionality to TTCN-3 while still remaining abstract and independent of actual implementation. Libraries for common constructs like stacks, collections, tables can be defined or automatic import mechanisms could be provided.

If an object of an external class is instantiated, it implicitly creates an external object and the internal object has a handle to the external one. The reference to the external object is called a handle. When an external method is invoked on the internal object, the call is delegated to the handle.

NOTE 2: External objects are possibly shared between different parts of the test system. Therefore, racing conditions and deadlocks have to be avoided by the external implementation.

An internal class can extend an external class and add internal behaviour, but also additional external functions, which have to be declared external explicitly. Such a class is conceptually still an external class and each instance has a handle to an external object.

Restrictions

a) External classes shall not contain fields or functions with a body.

b) External classes shall not be derived from non-external classes other than object.

c) If an internal class defines an external function, it shall be derived from an external class either directly or indirectly.

EXAMPLE:

```
external type class Stack {
  function push(integer v);
  function pop() return integer;
  function isEmpty() return boolean;
}
```

5.1.1.3 Final Classes

If a class shall not be subclassed, it may be declared as `@final`. Final classes cannot be abstract.

5.1.1.4 Constructors

Syntactic Structure

```
create "(" { FormalParameter , }* ")"
[":" SuperClass "(" { ActualParameter , }+ ")" ]
StatementBlock
```

Semantic Description

A class can define a constructor called `create`. If no constructor is defined, a default constructor is implicitly provided where the formal parameters of the constructor are the parameters of the (implicit or explicit) constructor of the direct superclass and one formal in parameter for each declared member field of the class itself in their order of declaration with equivalent type. The constructor is invoked on a type reference to the class and the result of this invocation is a new instance object of the constructor's specific class. If a class is extending another class with an explicit constructor, that constructor shall be invoked by adding a super-constructor clause with an actual parameter list to the constructor declaration. An implicit constructor will automatically pass the required actual parameters to the constructor of its superclass.

In the constructor, it is allowed to refer to the object being constructed as `this` to reference the fields of the object to be created in case that the names of the formal parameters clash with the names of those fields. They are explicitly allowed to have the same names as class members.

EXAMPLE:

```
type class MyClass {
    var integer a;
    const float b;
    // implicit constructor:
    //create(integer a, float b) {
    //  this.a := a;
    //  this.b := b
    //}
}

type class MyClass2 extends MyClass {
    template integer t;
    // explicit constructor
    create(template integer t) : MyClass(2, 0.5) {
        this.t := t;
    }
}
```


5.1.1.5 Destructors

Syntactic Structure

```
finally StatementBlock
```

Semantic Description

A destructor may be provided using a finally declaration following the class body. This destructor will be invoked automatically at the latest before the system deallocates an object instance (which is tool specific and out of the scope of the present document) or when the owning component is terminates. The *StatementBlock* has access to all members accessible to the class. The *StatementBlock* is semantically a function body of a function without return clause.

When deallocating the object instance, the destructor of the associated class is invoked first, followed by the destructor of all parent classes in the reverse order of superclass hierarchy.

5.1.1.6 Methods

A method is a function defined inside the class body. It has the same properties and restrictions as any normal function, but it is invoked in an object which can be referred to by the `this` object reference. A method invocation can access the class's own fields and also the inherited protected fields and methods of its superclasses.

A method inherited from a superclass can be overridden by the subclass by redefining a function of the same name and with the same formal parameter list. When a method is called in an object, the version of the most specific class of the super class hierarchy of the concrete class that defines the method in its body will be invoked. The overridden method can be invoked from the overriding class by using the keyword `super` as the object reference of the invocation. If a method shall not be overridden by any subclass, it can be declared as `@final`.

Public methods, if not overridden by the subclass, are inherited from the superclasses. If a public method is declared in a class, it can be invoked also in all objects of its direct or indirect subclasses.

If a public method is overridden, the overriding method shall have the same formal parameters in the same order as the overridden method. Public methods shall be overridden only by public methods. Protected methods may be overridden by public or protected methods.

The return type of an overriding function shall be the same as the return type of the overridden function with the same template restrictions and modifiers.

Methods shall have no `runs on`, `system` or `mtc` clause directly attached to them. However, they inherit these clauses from their surrounding class.

5.1.1.7 Method invocation

Syntactical Structure

```
[ObjectInstance "."] Identifier "(" FunctionActualParList ")"
```

A method invocation is a function call associated with a certain object defined in the class of that object.

Methods are invoked using the dotted notation on an object reference. Inside the scope of a class, methods of the same class or any visible inherited methods can be invoked without the *ObjectInstance* prefix if the object the method shall be invoked in is the same object as the one invoking it. The usual restrictions on actual parameters, as well as runs on, mtc and system types apply also on method invocations. All other restrictions that apply to called functions also apply to method invocation.

5.1.1.8 Visibility

Fields can be declared as private or protected. Methods can be declared as private, public or protected. If no visibility is given then the default modifier protected is assumed.

Private member functions are not visible and can be present in multiple classes of the same hierarchy with different parameter lists and return values.

Public member functions can be called from any behaviour running on the object's owner component.

Restrictions

a) A field of any visibility can not be overridden by a subclass.

b) A public member function can only be overridden by another public member function.

c) Private members can only be accessed directly from inside their surrounding class's scope.

5.1.1.9 Built-in classes

The abstract special built-in class called `object` is the superclass for all classes that do not explicitly extend another class.

The pseudo definition of that class is:

```
type class @abstract @builtin object {
        // This function will return a tool-specific descriptive string by default
        // but can be overridden by subclasses
        public function toString() return universal charstring;
}
```

NOTE: The @builtin is only added for illustrative purposes and not part of the TTCN-3 language.