SLOVENSKI STANDARD
SIST ES 203 790 V1.4.1:2022

01-junij-2022

Metode za preskušanje in specificiranje (MTS) - 3. različica zapisa preskušanja in krmiljenja preskusov - Razširitev nabora jezikov TTCN-3: objektno orientirane lastnosti

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

Ta slovenski standard je istoveten z: ETSI ES 203 790 V1.4.1 (2022-04)

ICS:

35.060	Jeziki, ki se uporabljajo v informacijski tehniki in tehnologiji	Languages used in information technology

SIST ES 203 790 V1.4.1:2022 **en**

ETSI ES 203 790 V1.4.1 (2022-04)

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

Reference

RES/MTS-203790v141

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 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

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 prevailing version of an ETSI deliverable is the one made publicly available in PDF format at www.etsi.org/deliver.

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

If you find a security vulnerability in the present document, please report it through our Coordinated Vulnerability Disclosure Program:
https://www.etsi.org/standards/coordinated-vulnerability-disclosure

Notice of disclaimer & limitation of liability

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or other professional standard and applicable regulations.
No recommendation as to products and services or vendors is made or should be implied.
No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.
In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are 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 Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, 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.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

The use of underline (additional text) and strike through (deleted text) highlights the differences between base document and extended documents.

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, symbols 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 Symbols

Void.

3.3 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 object-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 discrimination 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 | @trait]
Identifier [extends ClassType {"," 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, timers, classes), methods and properties 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.

All fields may be declared without initializer, even const and template fields.

A class can be declared with the @trait modifier. Such a class is called a trait class. Other classes are called normal classes. A trait class is an abstract class and can not be instantiated. It also shall only declare methods without function bodies as members and no constructor.

A normal class can extend at most one other normal class and also any number of trait classes. The extended normal class is called the superclass, the extended trait classes are called the supertraits, while the extending class is called the subclass of all the classes it extends. Trait classes can only extend trait classes but not normal classes. 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 supertraits 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 inherits all members of its superclass and its supertraits that it does not override in its own class body. A non-private non-abstract member from the superclass can always be accessed inside the class body by using the dotted notation on the keyword **super**. Non overridden non-private members can be accessed without any dotted notation before the member name.

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.

Classes can be used as field or element types of structured types.

Restrictions

a) Void.

b) Passing of object references and structured types containing fields or elements of class type 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 structured types containing fields or elements of class type are allowed.

e) Class type cannot be the contained value of an any type 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 and the supertraits.

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

j) Class extension shall not contain cycles such that a class directly or indirectly extends itself.

k) Reference to a class shall not occur more than once in the list of classes being extended.

l) Neither fields not non-abstract methods shall be declared in trait classes.

m) Trait classes shall not define a constructor and shall not define a finally block.

n) A class shall extend at most one normal class.

o) If a structured type contains a field of a class type, this type is not seen as a data type and its values cannot be used for encoding or decoding, sending or receiving and neither used as an actual parameter (or part thereof) to a function started on another component.

Examples

EXAMPLE 1:

```
external function newGlobalId() return charstring;

type class @trait Identifiable {
    public function @abstract setId(charstring id);
    public function @abstract getId() return charstring;
}

type class MyIdentifiableClass extends Identifiable {
    create() {
        setId(newGlobalId());
    }

    var charstring id;
```

```
    public function setId(charstring id) { this.id := id }
    public function getId() return charstring { return id }
}

var Identifiable v_idObj := MyIdentifiableClass.create();
var charstring v_id := v_idObj.getId();
```

EXAMPLE 2: parallel inheritance

```
type class @trait A {
    function @abstract f();
}

type class @trait B {
    function @abstract f();
}

type class C extends A, B {
    // legal, as it inherits A.f() and B.f() and they have the same parameters and return clause
}

type class @trait B2 extends A {
    function @abstract f(); // overrides A.f()
}

type class C2 extends A, B2 { // legal, as B2 does not clash with A
    function f() { ... } // implements A.f() and B2.f()
}

type class C3 extends A {
    function f() { ... } // implements A.f()
}

type class D extends C2, C3 {
    // illegal, as it only one non-trait class can be inherited
}

type class E extends A, C2 {
    // legal, but inheriting A is redundant
}
```


5.1.1.1 Scope rules

Class constitutes a scope unit. For the uniqueness of identifiers, the rules specified in 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
```

```
        }
    }
}
```

5.1.1.2 Abstract classes

A class can be declared as @abstract. In that case, it is allowed that it also declares abstract member functions, abstract properties or properties with abstract getters or setters 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.

Abstract getters and setters have no body but the properties containing them can be referenced in all concrete instance of subclasses of the abstract class declaring them. Other members of the abstract class or its subclasses may reference abstract properties as if they were concrete. At runtime the concrete overriding definition will always 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, property getters and setters 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.

5.1.1.3 External classes

A class may also be declared as external. In that case, it may declare external member functions without a function body. It is allowed to omit the external keyword from these function declarations. External classes can extend non-external classes but classes not declared as external shall not extend from external classes. External classes may also define other members like normal classes. When instantiating an external class, the external object being created is provided by the platform adapter and the external method calls to the external 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 in 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.

Restrictions

a) Void

b) Void

c) Void

d) An internal class shall not extend an external class

EXAMPLE:

```
type class @abstract Collection {
  function @abstract size() return integer;
  // internal default implementation
  function isEmpty() return boolean {
    return size() == 0
  }
}
```

```
type external class Stack extends Collection {
  function push(integer v);
  function pop() return integer;
  function isEmpty() return boolean; // external implementation overrides internal
  function size() return integer; // external implementation of abstract function}
```

5.1.1.4 Final Classes

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

5.1.1.5 Constructors

Syntactic Structure

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

Semantic Description

A class may define a constructor called `create`.

If no constructor is defined inside a class body, an implicit default constructor is provided where the formal parameters of the constructor are the parameters of the (implicit or explicit) constructor of the direct superclass and one additional formal **`in`** parameter for each declared **`var`** and **var template** field or automatic property of the class itself unless they are declared with the **`@internal`** modifier and also all **`const`** or **`template`** fields with no initializer in their order of declaration with the same type as in the declaration. If a **var** or **var template** field has an initializer, the additional formal **in** parameter created for it, for the implicit constructor, shall have the initializer value as the default value of the formal parameter.

NOTE: Having a default value in the implicit constructor for the **var** and **var template** fields with initializer, makes it possible to skip that parameter when invoking the implicit constructor, or to override it with another value if explicitly provided.

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 a constructor with at least one parameter without default, that constructor shall be invoked by adding a super-constructor clause to the constructor declaration. The super-constructor clause consist of a reference to the class being extended and an actual parameter list. 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.

When an object is created via the invocation of a constructor, the fields of each class body in the class hierarchy that have initializers are initialized before the execution of that class body's constructor body. The fields of a superclass that have initializers are initialized before the fields of the subclass. Also, the constructor of the superclass is executed before the constructor body of the subclass. Thus, it is ensured that all initialization of the superclass hierarchy as well as local fields with initializers is finished before the execution of a constructor body.

Since the members of a class body can appear in any order and forward references are allowed between them, a field with an initializer which is referenced by the initializer of another field, is initialized first.

As the underlying external constructor of external classes might need additional parameters, these can be provided via the additional external formal parameter list. If no internal constructor needs to be defined, the constructor may be defined without external formal parameter list and no body. In that case, the formal parameter list defines the formal parameters passed to the external constructor.

Restrictions

a) All formal parameters of the constructor shall be **`in`** parameters.

b) The constructor body shall not assign anything to variables that are not local to the constructor body or accessible fields of the class the constructor belongs to.

c) The constructor body shall not use blocking operations.

d) The initialization of a member field shall not invoke any member function in the object being initialized.

e) The constructor body shall not invoke any member function in the object being initialized.

f) A member constant or template shall be initialized exactly once, either by its initialization part or by at most one constructor body.

g) Direct or indirect cyclic initialization is not allowed. That is the initializer of a field shall not use the same field directly or indirectly.

h) The initializer of a field shall not use a field that does not have an initializer.

EXAMPLE 1:

```
type class MyClass {
    var integer a;
    const float b;
    const float c := 7;
    template float myTemplate := ?;
    // implicit constructor:
    // only using variable fields and non-variable fields with no initializer
    //create(integer a, float b) { // no parameter for c and myTemplate
    //  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;
    }
}

type class MyClass3 extends MyClass {
    var float f;
    // implicit constructor:
    // create(integer a, float b, float f) : MyClass(a, b) {
    //   this.f := f;
    // }
}
```


EXAMPLE 2:

For each initialization statement it is marked with its initialization order in the comment.

```
type class MySuperClass {
    var integer a := 5; // 1
    const float b;
    create(integer a, float b) {
        this.a := a; // 3
        this.b := b; // 4
    }
}

type class MySubClass extends MySuperClass {
    var template integer t := ?; // 2
    create(template integer t) : MySuperClass(2, 0.5) {
        this.t := t; // 5
    }
}
```

EXAMPLE 3:

```
type class MySuperClass {
    var integer a := 1;
    var float b;
    // implicit constructor:
    // only using variable fields with and without initializer
    //create(integer a := 1, float b) {
    //  this.a := a;
    //  this.b := b
```