TTCN-3 Language Extensions
Object-Oriented Features
(ETSI ES 203 790)

Presented by: Axel Rennoch, Jens Grabowski, György Réthy, Kristóf Szabados, Tomas Urban, Jacob Wieland, Philip Makedonski

For: TTCN-3 Webinar

09.10.2020
Agenda

- Motivation and ideas
  - Added value of OO for TTCN-3
  - General introduction on OO
  - Differences against OO programming languages

- Details on TTCN-3 OO languages features
  - Definition of class types using methods and fields
  - Exception handling
Object Orientation:
Motivation and ideas
Object Orientation: Motivation and added value

✔ Heighten appeal of TTCN-3 to users used to object-oriented programming
✔ Use advantages of object-oriented modelling
✔ Reduce TTCN-3 emulation of object-oriented features
✔ Allow simple access to external objects

✔ Handling of larger and more complex tests (hiding of "local" details)
  ✔ Data and functions operating on it are kept together
  ✔ Providing support for fine-grained information hiding
  ✔ More attractive for OO software developers
General introduction to OO programming

✔ Concept of objects which can contain data (fields: attributes or properties) and behaviour (procedures: methods)

✔ Object’s procedures can access and often modify the data fields of the objects („this“ or „self“)

✔ OO programs are designed out of objects that interact with each other

✔ Most OOP languages are class-based, i.e. objects are instances of classes (their „types“)
Differences against common OO programming languages

✔ Methods can be overridden, but *not* overloaded. Private members can *not* be overridden.
   ⇒ Less confusion

✔ Fields can *not* be public
   ⇒ Local data responsibility

✔ No multiple or interface inheritance (so far)
   ⇒ Avoid usual problems with multiple inheritance and name-clashes

✔ No static members
   ⇒ TTCN-3 does not allow global variables
   ⇒ Instead of static functions, global functions can be used
Differences against common OO programming languages – New Features

✔ Objects are *owned* by the component creating them. Methods can only be called by behaviour running on the owning component.
  ➞ No data racing conditions

✔ **Classes can have runs on, mtc and system clauses,** restricting test system context and usage of classes
  ➞ No repetition of clauses for methods
  ➞ Statically checkable safe access to test system environment

✔ External classes
  ➞ Simple access to external objects

✔ Implicit constructor
  ➞ Less boilerplate code
Definition of class types
A **class** defines a new **TTCN-3 type**, containing one or more members:

- **Fields**: var, const, template, port, timer
- **Methods**: function, constructor (create)

```tcl
type class Person {
  //fields
  var charstring v_name;

  //methods
  function f_nameLog() {log(v_name);...};
  create() {this.v_name := "noname";}
}
```

Reference a member of own object (class)
Objects of classes (1)

type class Person {
  var charstring v_name;
  function f_nameLog() { log(v_name); ...};
  create() { this.v_name := "noname"; }
}

✔ An **object** is an instance (i.e. a **value**) of a **class**,
  ✔ comprising a data instance of each field of the class,
  ✔ created after invocation of the constructor of the class
  ✔ can be created in a behavior running on a TTCN-3 component (the **owner** of the object)
Objects of classes (2)

Type class Person {
    var charstring v_name;
    function f_nameLog() { log(v_name); …};
}

Implicit constructor (*not required to be specified*):
create (charstring v_name) { this.v_name := v_name; }

An object reference is contained in a variable of a class type.

Creation and use of the object

var Person v_chair := Person.create("Anthony");

v_chair.f_nameLog() 

output is: Anthony
Inheritance of classes

A class definition inherits all declarations from its super class

```plaintext
type class Person {
    var charstring v_name;
    function f_nameLog() {log(v_name);...};
}

type class EtsiPerson extends Person {
    var charstring v_position;
}
```

✔️ A class definition inherits all declarations from its super class
Visibility of members

✔ Fields are private or protected (default is protected)

✔ and can not be overwritten

type class Person {
    var charstring v_name;
    function f_nameLog(charstring p_c) {log(v_name);...};
}

type class EtsiPerson extends Person {
    var charstring v_position;
    public function f_nameLog(charstring p_c) {log(v_name & p_c);...};
}

✔ private and protected members can not be accessed outside their class

var EtsiPerson v_chair := EtsiPerson.create("Anthony", "Chair");
v_chair.f_nameLog("56");
log(v_chair.v_name);

ERROR: no access to fields

output is:
Anthony56
Visibility of members (cont.)

- **Methods** are *private, protected* or *public* (default is protected)

  ```
  type class Person {
    var charstring v_name := "Anthony";
    function f_nameLog(charstring p_c) {log(v_name);...};
  }
  
  type class NewPerson extends Person {
    function f_nameLog(charstring p_c) {log(v_name & p_c);...};
    public function f_superLog() {super.f_nameLog("Alex");};
  }
  ```

- **Public** member functions can only be overwritten by *public* member functions and can be called from any behavior on the object’s owner component

  ```
  var EtsiPerson v_chair := NewPerson.create;
  v_chair.f_superLog();
  ```

  output is: *Anthony*
Component type restrictions

健全 runs on, system, mtc clauses restrict the component context that can create objects of that class and call methods of the class (if missing, inherited from superclass) and shall be compatible with superclass clauses

健全 function members inherit restrictions from the containing class (no own runs on, system, mts clauses)

type component MyComponent {
   port myport MyPortType;...
}

type class Person runs on MyComponent system MySUT mtc MyTester {
   var charstring v_name;
   function f_nameLog() {myport.receive;...};
}
Object reference and class casting

✔ To access an object instance an object reference is needed.

✔ The object is not copied when used as an actual parameter or assigned to a variable (only the reference).
  ✔ Multiple variables can contain a reference to the same object simultaneously.

✔ Objects cannot be shared by multiple components.

✔ Object references can be cast to another class
  ✔ New class shall be within the set of (direct or indirect) superclass or subclass

```plaintext
var Person v_person := EtsiPerson.create("Anthony");
var EtsiPerson v_etsichair := v_person => EtsiPerson;
```

Two references for the one object
Class type discrimination

- **of-operator** checks if most specific class of the object \( \text{left-hand} \) side is equal or subclass derived from the **class type** \( \text{right-hand} \) side.

- **select class-statement** discriminates the class of an object (allows superclasses and subclasses of the object).

```plaintext
testcase TC_1() {
    var MyClass v_a := MyClass.create;
    var MyClass v_b := MyClassB.create;
    if (v_a of MyClass) {...};
    select class (v_b) {
        case(MyClassB) {...}
        case(MyClassA) {...}
    }
}
```

**will be chosen**

**will not be chosen**
Outlook – Additional Features

- External classes and methods
- Abstract classes and methods, final classes

Already in the last standard version (2020):
- Nested classes
- Generic classes and methods
- Mixed classes (External classes with internal additional behaviour/state)

Next standard version (2021):
- Interfaces and multiple interface inheritance (similar to Java)
- Properties (similar to C#)

Additional Ideas Welcome!
Application example
Application example

✓ Application background:
  ✓ oneM2M common service/application elements (CSE/AE)
  ✓ sample use cases

✓ Specification of semantic descriptor (TTCN-3)
  ✓ class type instead of record type
  ✓ sample application (extended class)
Application background

✔ oneM2M common service/application elements (CSE/AE)

✔ Addition of semantics annotations:
   to discover dedicated AE’s (e.g. sensors), based on their location (e.g. area) or kind (e.g. temperature) etc.

✔ Possible scenarios:
   ✔ creation of AE representation at CSE (e.g. container, contentInstance), e.g. temperature sensors
   ✔ addition of semantic descriptors to AE representation, by other AE (e.g. dashboard)
   ✔ semantic discovery, requested by other AE (e.g. mobile handheld)
Semantic annotation

AE: Application Entity
CSE: Common Services Entity

Source: oneM2M.org
**SemanticDescriptor** in oneM2M

```plaintext
type record SemanticDescriptor {
    ResourceName resourceName,
    ResourceType resourceType,
    XSD.ID resourceID,
    NhURI parentID,
    Timestamp creationTime,
    Timestamp lastModifiedTime,
    Labels labels optional,
    AcpType accessControlPolicyIDs optional,
    Timestamp expirationTime,
    ...
}
```

- ✔️ Currently (for historical reasons) using `record type` for `SemanticDescriptor`
- ✔️ For the sake of simplicity the example leaves out some fields
- ✔️ Note: related behavior (such as field set/get functions) is defined separately
**SemanticDescriptor using class type**

- Introduction of **class type** for **SemanticDescriptor**

- Additional class fields can be provided if using class **inheritance**
OO application example

Extension of SemanticDescriptor for simplified handling of context-related details

Example:
TemperatureSemanticDescriptor extends SemanticDescriptor

- Add context information, e.g. temperature types (C/F), usage (indoor/outdoor), manufacture information (country, price etc.), temperature ranges (-30...40), defaults/standards
- Add related functionality (translation formula, exchange rates): f_compatibility, f_translate
type class TemperatureSemanticDescriptor extends SemanticDescriptor {
  var charstring MeasurementUnit;
  var charstring Usage;
  var integer LowerLimit;
  var integer UpperLimit;
  function f_compatibility(...) return boolean;
  function f_translate(...);
  create (charstring p_mu, charstring p_u, integer p_ll, integer p_ul):
    SemanticDescriptor(...)
    {this.MeasurementUnit:= p_mu, this.Usage:= p_u,
     this.LowerLimit:= p_ll, this.UpperLimit:= p_ul,};
}

var TemperatureSemanticDescriptor v_tSensorEU :=
  TemperatureSemanticDescriptor.create("celsius", "outdoor", -30, 40);

var TemperatureSemanticDescriptor v_tSensorUS :=
  TemperatureSemanticDescriptor.create("fahrenheit", "indoor", 32, 104);
Exception handling
Exception handling

- Exception type lists: functions, external functions, altsteps
- \textit{raise} exception statements
- "catch" and "finally" clauses: statement blocks, altsteps and testcase
**raise** Exception statement

✔️ Causes **leaving** of: statement block, loop, alt, interleave
  ✔️ within the encompassing function/altstep/testcase

(1) Execution **continues** in the *catch-block*
  ✔️ If encompassing function/altstep/testcase has *catch-block* (with same type, or can be cast)

(2) Execution **leaves** function/altstep/testcase
  ✔️ If **NO catch block** available or can **handle** the raised exception
  ➢ Handle the exception in the **calling** function/altstep/testcase

✔️ **Dynamic error**, if exception not handled at the latest *catch* clause of the testcase statement block
Exception handling samples

(1) execution continues in catch-block

```plaintext
function f_myf1() exception (integer) {...
    raise integer:1;
} catch (integer p_i) {...}
```

(2) execution continues outside

```plaintext
function f_myf1() exception (integer) {...
    raise integer:1;
}
```

Do something!
Exception handling – application example

✔ Simplification of post processing in case of error handling
✔ E.g. resource creation and/or resource releases
✔ Initialization scenario based on sub-processes for registration and request message
function f_create(in charstring p_name) exception (charstring, integer) 
    runs on myComponent
{
    var integer v_rc:= -1;
    ...
    if (not fx_register(p_name)) {
        raise("Could not register" & p_name);
    }
    ...
    if (not f_sendRequest(p_name, v_rc)) {
        raise(v_rc+1000);
    }
    ...
}

catch (charstring p_c)
{log("Initialization failed: ", p_c); setverdict(inconc)...}

catch (integer p_i)
{log("Creation failed with return code: ", p_i); setverdict(fail);...}
Conclusions
Key takeaways

ække

Enhancements for TTCN-3 programmers

Attraction of new users

Ongoing maintenance and improvements by ETSI TTF
Q&A

For further information please visit www.ttcn-3.org
and/or contact ETSI TC MTS via www.etsi.org/MTS.

Team: https://portal.etsi.org/STF/STFs/STF-HomePages/T003
Community: http://www.ttcn-3.org/index.php/community/contact
Suggestions: http://www.ttcn-3.org/index.php/community/change-requests
Short summary on TTCN-3 in general (Webinar part 1)

✔ Abstract test definition language
  ✔ Used in multiple industrial domains
  ✔ One testing technology for all testing types:
    functional (conformance, functions) and non-functional (performance, security, vulnerability)

✔ Long history in standardization (ISO, ITU-T and ETSI)

✔ Independent from programming languages
  ✔ Includes testing specific features
  ✔ Mappings to Java, C++, C#
  ✔ Extensibility via attributes, external functions etc.
  ✔ Integration with different languages like JSON, XML, ASN.1, IDL

✔ Earlier versions were lacking modern OO features
Test component and port concepts of TTCN-3

✔ Test components are independent entities
  ✔ Each one is running a piece of the whole test case behaviour...
  ✔ and owning its own data and objects
  ✔ MTC – Main test component is created automatically
  ✔ PTC – Parallel test component(s) created dynamically

✔ Communicating with each other and with the SUT via ports
  ✔ Defined sets of incoming & outgoing messages, and procedure calls & responses