



EXtended **A**utomation **M**ethod

Concept paper

Version:	3.0
Date:	29. July 2009
Status:	Approved
Author:	Gerhard Kiffe (AUDI AG Ingolstadt / Abt. I/EE-65)
File:	EXAM_concept_paper_en.doc




Concept paper	EXAM EXtended Automation Method	Department: I/EE-65	
		Page: 2 of 104	
		Revision date: 7/29/2009	

Table of contents:

1	HISTORY	4
2	GLOSSARY	5
2.1	ABBREVIATIONS.....	5
2.2	GENERAL TERMS	6
2.3	UML CONCEPTS.....	7
3	PREFACE	11
4	EXAM PROCESS MODEL [PM]	12
4.1.1	Layer IV.....	12
4.1.2	Layer V.....	15
4.1.3	Layer VI.I.....	17
4.1.4	Layer VI.II.....	19
5	EXAM PROCESS MAPPING	21
5.1	TESTDESIGN.....	22
5.1.1	SpecificationLayer	22
5.1.1.1	Terms [EXAM MM]	22
5.1.1.2	Relation between terms form the EXAM meta model [EXAM MM]	25
5.1.1.3	Modeling	26
5.1.2	ImplementationLayer	48
5.1.2.1	Terms [EXAM MM]	48
5.1.2.2	Relation between terms form the EXAM meta model [EXAM MM]	49
5.1.2.3	Modeling	50
5.1.3	CompositionLayer.....	55
5.1.3.1	Terms [EXAM MM]	55
5.1.3.2	Relation between terms form the EXAM meta model [EXAM MM]	55
5.1.3.3	Modeling	56
5.1.4	ExecutionLayer	57
5.1.4.1	Terms [EXAM MM]	57
5.1.4.2	Relation between terms form the EXAM meta model [EXAM MM]	58
5.1.4.3	Modeling	59
5.2	TESTOUTPUT	62
5.2.1	Data model for result data	62
5.2.2	Definition of entities and attributes.....	63
5.2.3	Test valuation in EXAM	67
6	EXAM CONVENTIONS.....	68


Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	3 of 104	
		Revision date:	7/29/2009	

6.1	EXAM DATA TYPES	68
6.2	CONTROL FLOW ELEMENTS IN EXAM	71
6.2.1	Control flow elements in TestSequences	72
6.2.2	Control flow elements in TestActivities	74
6.3	EXPRESSIONS IN EXAM	75
6.4	IMPLICIT MEMORY AREAS	79
6.5	(MULTIPLE) INHERITANCE	80
6.5.1	Inheritance between ParameterSets	81
6.5.2	Inheritance of attributes	83
6.5.3	Inheritance of operations	86
6.6	OPERATION OVERLOADING.....	89
6.7	COMMENTS	90
6.8	SIGNATURE	90
6.9	NAMING CONVENTIONS	91
6.10	CALLER CLASS.....	91
6.11	DEVIATIONS FROM THE UML STANDARD	92
6.11.1	Including TestFlows in sequence diagrams.....	92
6.11.2	Signatures of TestFlow includes.....	93
6.11.3	Template concept for TestActivities.....	94
6.11.4	Indices in TestActivities	97
6.11.5	Synchronization in TestActivities	98
6.11.6	InterruptibleActions	100
7	EXAM LAYER MODEL.....	102
8	BIBLIOGRAPHY	104

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	4 of 104	
		Revision date:	7/29/2009	

1 History


Version:	Change:	Author (Name, OU):
3.0	Translation of the German version 3.0 by Gerhard Kiffe (AUDI AG I/EE-33)	Tim Warode (MicroNova AG)

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	5 of 104	
		Revision date:	7/29/2009	

2 Glossary

2.1 Abbreviations

Abbreviation:	Meaning:
ARIS	Architecture of Integrated Information Systems
ASCII	American Standard Code for Information Interchange
CASE	Computer Aided Software Engineering
ECU	Electronic Control Unit
EXAM	EXtended Automation Method
GUID	Globally Unique Identifier
ITS	Integrated Test Services
MDA	Model Driven Architecture
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
SUT	System Under Test
UML	Unified Modeling Language
UUID	Universally Unique Identifier

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	6 of 104	
		Revision date:	7/29/2009	


2.2 General terms

Term:	Description:
ARIS	The ARIS concept offers methods to design, implement and optimize integrated information systems. It is based on the economic process chains of a company. Particular business processes and their relations can be displayed using various model types, e.g. extended event-driven process chains, extended entity-relationship diagrams, function allocation diagrams or organization charts.
CASE-Tool	Software tools to support “Computer Aided Software Engineering“. Specification, design and implementation are supported using graphical methods (UML modeling, generation of application code). [ITS]
ECU	Electronic control device
EXAM	EXAM – EXtended Automation Method is a project to define and implement a test automation method for test benches that is standardized throughout the Volkswagen Group.
GUID	A Globally Unique Identifier (GUID) that is used within distributed computer systems.
MDA	Model Driven Architecture – an OMG standard that aims for a software architecture that separates business and application logic from the platform. [ITS]
OMG	The Object Management Group (OMG) is a consortium that has been founded in 1989. It mainly focuses on modeling (programs, systems and business processes) and defining model-based standards.
Repository	Schema to store connected formal EXAM test specifications within a database.
SUT	System under Test.


2.3 UML concepts

This chapter explains UML concepts that are relevant for EXAM.


Term:	Description:
abstract class	An abstract class is never instantiated. It is incomplete by purpose and serves as a basis for subclasses. [UML2.0]
abstract operation	An operation that is defined but not implemented. It is defined in an abstract class and implemented by concrete subclasses. [UML2.0] Classes containing abstract operations are abstract classes.
activity diagram	UML diagram type. Activity diagrams model complex processes using concurrence and branching. The diagram describes the flow of actions that are combined within activities. The UML activity diagram is based on the Petri net token concept.
action	An action is an atomic routine to specify the behavior of a system within an activity diagram.
activity	An activity describes a flow. It contains various types of nodes that are connected using object- and control flows. [UML2.0] The concepts “activity” and “activity diagram” are used synonymously.
association	An association describes a relation between classes, i.e. the common semantics and structure of a set of object relations. There are directed associations (only accessible from one side) and bidirectional associations (accessible from both sides). [UML2.0] associations are essential to ensure that objects can communicate with each other.
attribute	An attribute is a structural feature of a specified class and thus a part of the structure of instances of that class. It represents a data element and is identified by its name and type.
class	A class is a type. It has a number of characteristics, especially attributes and operations.
class diagram	UML diagram for the representation of classes and their relationships (inheritance, association, aggregation, dependency). [ITS]
decision	Display element in activity diagrams for the branching of the control flow; splitting the control flow into several alternatives (multiple outgoing edges). Every edge must have a condition.
dependency	A dependency is a relationship between two model elements, which shows that a change in one (independent) element requires a change in the other (dependent) element. [UML2.0]
edge	Edges are directed connections between the representation elements of activities and activity diagrams.

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	8 of 104	
		Revision date:	7/29/2009	


Term:	Description:
end node	The end node marks the end of a flow in an activity diagram.
fork	Display element in activity diagrams for the parallelization of the control flow; starting point of parallel actions.
generalization (specialization)	In UML, a generalization is a directed relationship between a general and a specific element. Concretely, this means that the special element implicitly has all the characteristics (structural and behavioral characteristics) of the general element. By implication, because these characteristics are not explicitly declared in the specific element. These characteristics are 'inherited' from the general element.
inheritance	Inheritance is a concept in object orientation. A derived object (e.g. class or use case) "inherits" the abilities and characteristics of a base object. Thus, the derived object is able to use the existing program logic of the base object.
instance	See object
interaction frame	New description element of the UML 2.0; Interaction Frames provide a context or area where diagrams or diagram elements can be created that make use of the special semantics of the frame. They are used to model control structures (e.g. loops) or references to external sequences in sequence diagrams. Interaction frames are depicted as rectangular boxes.
interface	Interfaces define operation signatures to form a public interface for model elements (mainly classes and components). [UML2.0] Interfaces can be implemented by classes.
implementation dependency	An implementation dependency is a directed dependency relationship with the «realize» or «implement» stereotype assigned. It connects a class with an interface.
join	Presentation element within activity diagrams to synchronize the control flow; endpoint of parallel actions. All parallel actions must be completed before the following actions can be executed.
lifeline	A lifeline is a display element in sequence diagrams that describes an object and its lifetime in the context of the communication process. It is used for interaction. An interaction can have multiple lifelines; but one lifeline can only belong to one interaction.
merge	Display element in activity diagrams to merge the control flow; merges alternative processes of the control flow. Unlike the synchronization node, no synchronization is done.
message	Messages allow objects to communicate with each other. A message provides information on which activity an object is expected to perform, i.e. a message asks an object to execute an operation. [UML2.0]
note	Comments or annotations to a chart, or one or more model elements without any semantic effect. [UML2.0]

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	9 of 104	
		Revision date:	7/29/2009	

Term:	Description:
object	An object is an existing and operating entity within a running system. Each object is an instance of a class. An object contains information represented by attributes, whose structure is defined in the class. An object can receive messages, i.e., has corresponding operations for every defined message. The behavior defined by the messages and the structure of its attributes is the same for all objects of that class. However, the values of the attributes may differ for individual objects. [UML2.0]
object diagram	An object diagram has a similar structure as the class diagram, but instead of classes, it contains an exemplary selection of existing objects and their particular values at a given point in time. [UML2.0]
operation	Operations are services provided by an object and may be requested using messages. Operations are described by their signature (operation name, parameters and return value (if applicable)). [UML2.0]
package	Packages are collections of model elements of any type. They can be used to divide the overall model into smaller, more manageable units. A package defines a namespace, i.e. names of the elements contained in a package must be unique. Each model element can be referenced from multiple packages; however, it belongs to one and only one (home-) package. Packages can include other packages. The topmost package includes the entire model. [UML2.0]
sequence diagram	UML diagram. Sequence diagrams are the most important interaction diagrams. They model a series of messages (operation calls) between objects over a specific period. This may include creating and removing objects.
signature	The signature of an operation consists of the name of the operation, its parameter list and possibly a return type. [UML2.0]
start node	The start node is the starting point of a process in an activity diagram.
stereotype	Category that can be used to classify UML artifacts. One UML artifact can have any number of stereotypes. Stereotypes and its names can be defined freely. [ITS]
step	Step within a sequence diagram.
subclass	A subclass is the specialization of a superclass and inherits all the properties of that superclass. [UML2.0]
superclass	A superclass is a generalization of selected properties of a subclass. [UML2.0]
tag definition	Property definition for stereotypes
tagged value	Tagged values are custom, language- and tool-specific key words. They extend the semantics of individual model elements by special characteristics. [UML2.0]

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	10 of 104	
		Revision date:	7/29/2009	

Term:	Description:
UML	Standardized graphical notation for object-oriented analysis, design and development of software systems. [ITS]
UML model	Collection of presentation elements in the UML standard definition (classes, relations, diagrams, etc.) that describe a self-contained object-oriented system. [ITS]
UML artifact	Element from an UML model (e.g. a class, relation or diagram). [ITS]
use case	A use case describes a time-continuous interaction (a single operation) of one or more stakeholders with a system from the user perspective.
use case diagram	A use-case diagram displays relations between stakeholders and use cases from the perspective of that stakeholder. Use Case diagrams are primarily used to model requirements; less often they are used for behavioral modeling and system design.
use case include	Use case within another use case, depicted by dashed arrows. «include» relation between use cases.


Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	11 of 104	
		Revision date:	7/29/2009	

3 Preface

This document provides a complete description of the EXtended Automation Method (EXAM). It defines all relevant terms, rolls and processes and their mapping to the Unified Modeling Language (UML). It forms the theoretical basis for a concrete implementation of EXAM.

In principle, EXAM is a comprehensive methodology for presentation, implementation and evaluation of test cases. The aim of EXAM is to formally describe test cases in UML models to make them machine interpretable and partly or fully transferrable into executable test programs. This allows for sharing and reusing test cases and –fragments – even if the actual test system differs. EXAM, in a broader sense, follows the Model Driven Architecture (MDA) approach.

MDA is, as well as UML, a standard of the Object Management Group (OMG). The objective of MDA is to model business and application logic of a software system platform-independently using UML. Platform specific code can then be generated automatically from the model. The MDA standard mainly focuses on the modeling of business processes and management software, regardless of the underlying platform. Such a Design is called PIM (Platform Independent Model). Configurations allow transforming the PIMs into PSMs (Platform Specific models). A development environment that follows the MDA standard is able to generate platform specific code from a PIM and its configuration. Developers have to deal with the platform-dependent aspects of their software only in these configurations. The MDA approach significantly increases the productivity and quality of the development process and assures the sustainability of development results.

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	12 of 104	
		Revision date:	7/29/2009	

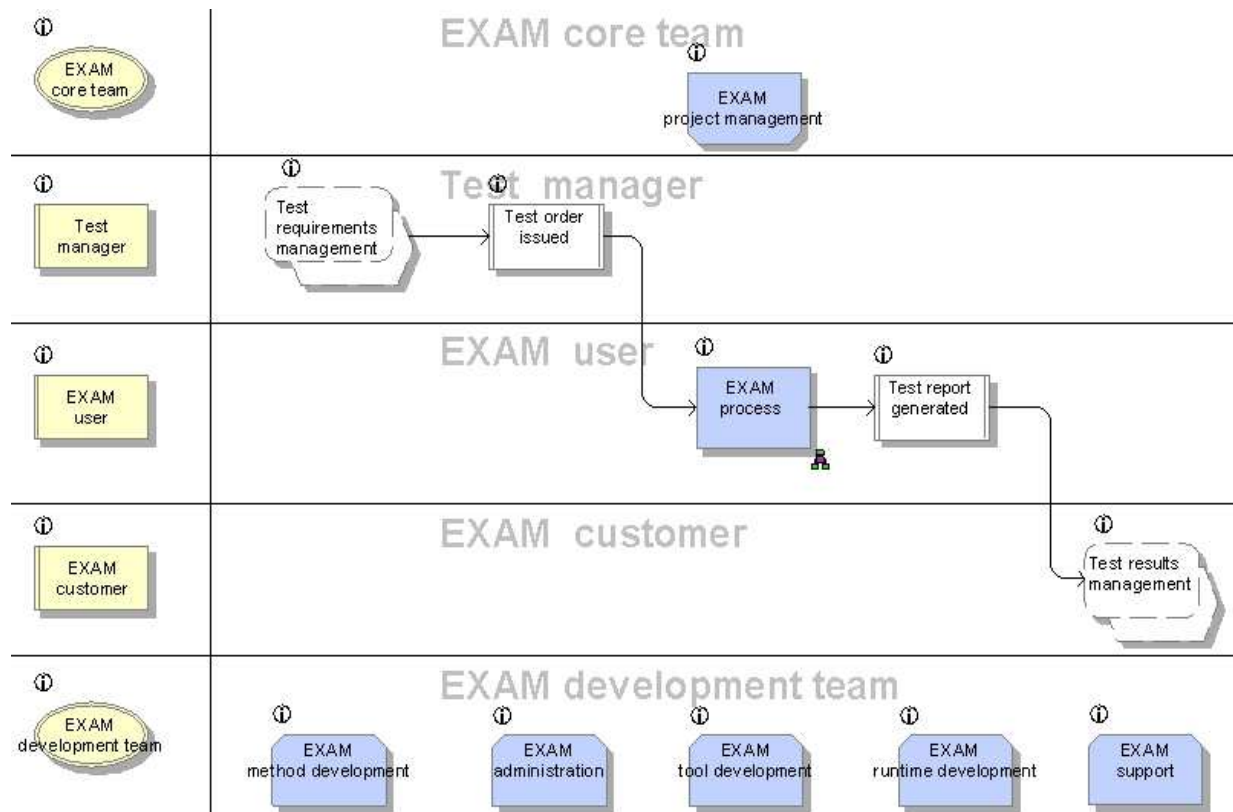
4 EXAM process model [PM]

The development of EXAM was based on the process model [PM] described in this chapter. The modeling of tests in EXAM is derived from the various process steps and implemented using appropriate software tools. The figures in this chapter follow the conventions described in the "Konventionenhandbuch zur Prozessmodellierung – Version 2.3"¹ [KONV]. According to these guidelines, modeling was done at layers 4 to 6.

EXAM processes are modeled using so-called Swim Lane diagrams. Each lane defines a solid competence which can either be a committee (yellow ellipse) or a role (yellow rectangle). Process steps are depicted in blue, business objects in white rectangles. The process chain is visualized using edges (arrows). In ARIS, every process gets a business object as input and delivers another business object as its output. Since the ARIS models are intuitively readable, we omit a more detailed description at this point.


4.1.1 Layer IV

The model on layer IV shows a general overview over the EXAM process.



Swim Lane diagram EXAM – layer IV


¹ Guide to process modeling, version 2.3

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	13 of 104	
		Revision date:	7/29/2009	

Committee	Description
EXAM core team	EXAM project management committee
EXAM development team	The EXAM development team consists of EXAM project leaders, method developers, administrators, tool- and runtime developers as well as EXAM supporters.

Role	Description
Test manager	The test manager is responsible for a test project. He controls which test cases are executed when and where. Often, he also evaluates test results and monitors changes to those results from different test runs. Thus, he may also be EXAM customer.
EXAM user	EXAM user is the generic term for all roles within the EXAM process.
EXAM customer	EXAM customers receive the results of the EXAM process, e.g. as a test report. Often, they are also the initiator/test manager of that particular test.

Process	Description
EXAM project management	The EXAM process management controls all activities within the EXAM project.
EXAM process	This process step represents the actual EXAM process. It is described in more detail in the lower process layers.
EXAM method development	The task of the EXAM method development is to evaluate promising scientific approaches and advanced technologies for their applicability within the EXAM project.
EXAM administration	The EXAM administration process provides the IT infrastructure of the EXAM project. This includes e.g. the application, code generator and database servers.
EXAM tool development	EXAM tool development involves development of all tools that are used by the EXAM users, e.g. modeling tools and assistants, generators and the test execution and evaluation tools.
EXAM runtime development	The EXAM runtime development provides basic methods to be able to execute tests on real test systems. This includes libraries to access e.g. simulation platforms, power supplies and data loggers as well as access and deployment mechanisms (e.g. Python, C++, Corba, etc.).
EXAM support	EXAM not only is an IT system, but a method. It is necessary to support EXAM users with their tasks and to assure compliance with the conventions and philosophies of EXAM as well as to drive the standardization process forward.

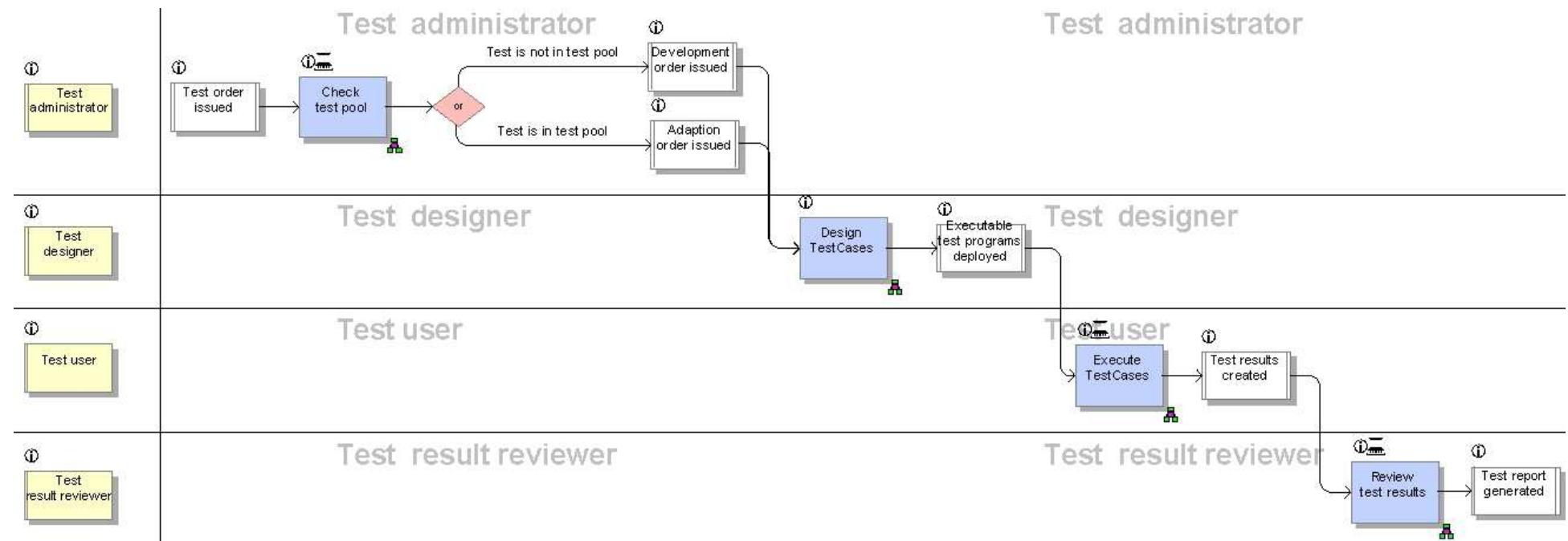
Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	14 of 104	
		Revision date:	7/29/2009	

Business object	Description
Test order issued	A test order is an official assignment. It requests execution of tests according to a test specification. The test specification is required for any test order and may or may not be formal.
Test report generated	The representation of the processed test results is called test report.


Process interface	Description
Test requirements management	The test requirements management process is responsible for defining the test scopes and their assignment. Test scopes are defined in a test specification. Its concrete form is dependent on many basic conditions and its detail varies widely. From the EXAM point of view, it is therefore called an informal test specification. The informal specification test is usually an annex to the test order. This process is not part of the EXAM approach.
Test results management	The test results management processes the results produced and qualified by EXAM. This can be done for example by an error tracking system. The official input document to this process is the test report, which can have various formats (e.g. PDF, XML, HTML, EXCEL, etc.). This process is not part of the EXAM approach.

4.1.2 Layer V

The model on layer V details the EXAM process.



Swim Lane diagram EXAM – layer V

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	16 of 104	
		Revision date:	7/29/2009	

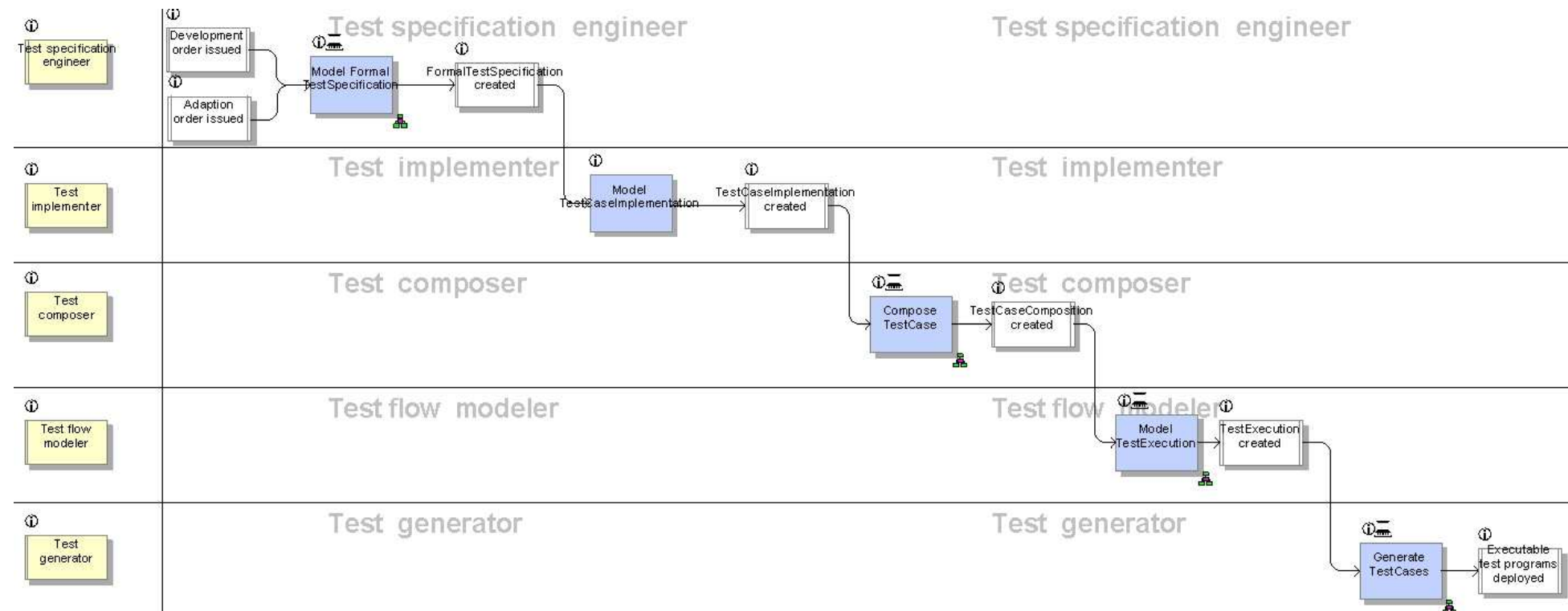
Role	Description
Test administrator	The test administrator manages the testing pool (all available test cases) or a part thereof. He has an overview of existing test cases and possibly publishes new test cases. In addition, he is responsible for the versioning and standardization of test cases.
Test designer	Generic term for all the roles in the field of test development activities.
Test user	The user test executes existing test cases and monitors the test run.
Test result reviewer	The test result reviewer analyzes the results produced during test execution and verifies them for correctness with regard to the EXAM flow. He documents problems. By definition, he does not carry out substantive/technical assessments of the results.

Process	Description
Check test pool	The test pool contains the entirety of available test cases. This process step checks whether there are already test cases for an existing, qualified test topic within the central repository (test sourcing).
Design TestCases	Process of modeling and generating executable test programs.
Execute TestCases	The designed test cases are executed on a test system according to their particular test specification.
Review test results	The process checks whether the results of a test run are free of errors. Note, that no assessments with regard to contents are done definition. This is the task of EXAM customers.


Business object	Description
Development order issued	If the required test cases are not contained in the test pool, they have to be <u>created newly</u> .
Adaption order issued	If the required test cases are contained in the test pool, they have to be <u>adapted</u> to the actual test subject (test system, SUT, etc.), for example by parameterization.
Executable test programs deployed	Executable test programs for a specific test subject have been deployed.
Test results created	The test results that have been created during test execution are ready for further processing.

4.1.3 Layer VI.I

The model on layer VI.I details the “Design test cases” process.




Swim Lane diagram EXAM – layer VI.I

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	18 of 104	
		Revision date:	7/29/2009	

Role	Description
Test specification engineer	The test specification engineer is responsible for modeling the formal test case in the UML. Thus he needs to be familiar with the test subject. He takes care to define specific characteristics of the test case abstract to ensure portability to other test systems or SUTs.
Test implementer	The test implementer creates the implementation classes for the library that is required to execute an abstract test case on a concrete test system. This can be done either in code or UML. In addition, he creates the system configurations for the mapping of formal test specification and implementation classes.
Test composer	The test composer creates unambiguous assignments of test flow and parameter sets, i.e. the calibration of the test cases.
Test flow modeler	The test flow modeler defines the test execution structure in the UML model. He uses test campaigns, test suites and test groups to configure the execution order and to separate competences within test projects.
Test generator	The test generator transforms the completely modeled test cases into executable test programs using the code generator.

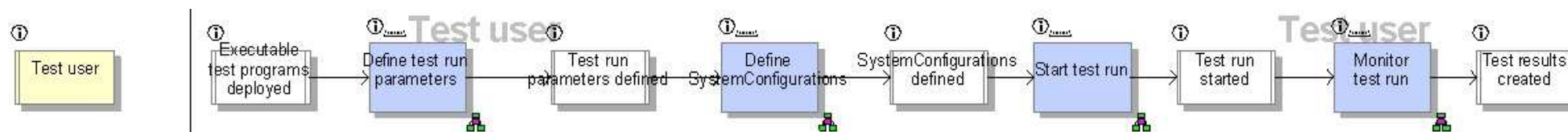
Process	Description
Model FormalTestSpecification	Modeling the formal test specification as UML model in EXAM using the available description items.
Model TestCaseImplementation	Modeling and programming the concrete implementation classes for the test subject or test system, and creating system configuration(s) containing assignments of formal test specification and implementation classes.
Compose TestCase	Creation of unambiguous assignments of test flow and parameter sets, i.e. the calibration of the test cases.
Model TestExecution	Definition of the test execution structure in the UML model using test campaigns, test suites and test groups to configure the execution order and to separate competences within test projects.
Generate TestCases	Transformation of the completely modeled test cases into executable test programs using the code generator.

Business model	Description
FormalTestSpecification created	A formal test specification in form of a UML model is available.
TestCaseImplementation created	All required implementation classes and system configurations are available in the UML model.
TestCaseComposition created	The concrete, unambiguous calibration of the test cases has been defined
TestExecution created	The structure of the test flow, consisting of test campaigns, test suites and test groups has been modeled correctly.


Concept paper	EXAM EXtended Automation Method	Department: I/EE-65	
		Page: 19 of 104	
		Revision date: 7/29/2009	

4.1.4 Layer VI.II

The model on layer VI.II details the “Execute test cases” process.



Swim Lane diagram EXAM – layer VI.II

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	20 of 104	
		Revision date:	7/29/2009	

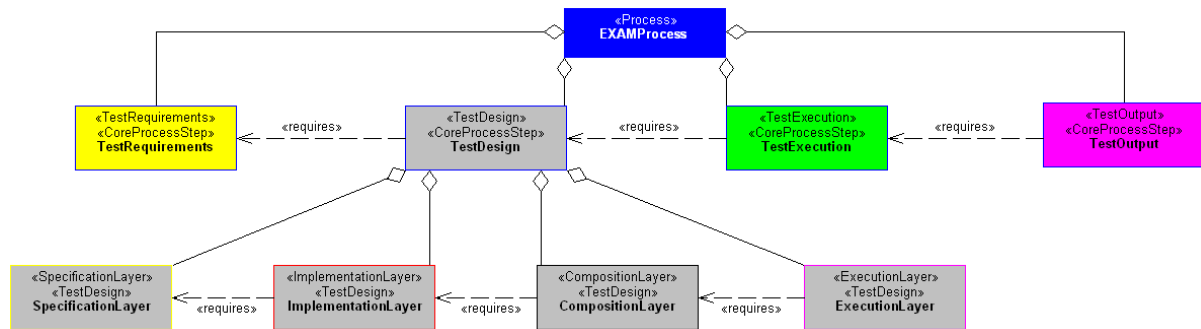
Role:	Description:
Test user	The test user executes available test cases and monitors the execution phase.

Process:	Description:
Define test run parameters	Configuring relevant parameters, e.g. where to store test results, log level, execution mode etc.
Define SystemConfigurations	Defining on which test system resp. with which configuration the test run is to be executed.
Start test run	Starting the test run.
Monitor test run	Monitoring the test run if required. May be beneficial to find and correct errors in the test flow.

Business object:	Description:
Test run parameters defined	All relevant parameters have been defined.
SystemConfigurations defined	The actual system configuration has been defined.
Test run started	Test execution has been started successfully.

5 EXAM Process mapping

This chapter provides a methodic description of relevant process steps in EXAM based on the EXAM process model which has been introduced in chapter 4.



Top level EXAM meta model [EXAM MM]

The figure shows the topmost layer of the EXAM meta model which has a well-defined mapping to the EXAM process model. The table below describes the correlation between the particular model parts.

EXAM process model	EXAM meta model
4.1.1 EXAM process	EXAMProcess
4.1.1 Test requirements management	TestRequirements
4.1.2 Design test cases	TestDesign
4.1.3 Model formal test specification	SpecificationLayer
4.1.3 Model test case implementation	ImplementationLayer
4.1.3 Compose test case	CompositionLayer
4.1.3 Model test execution	ExecutionLayer
4.1.2 Execute test cases	TestExecution
4.1.2 Review test results	TestOutput

The test requirements topic is not covered in this concept paper because it is not part of the EXAM methodology. The test execution is only handled from the modeling point of view, since the software implementation of the EXAM meta model and its objects and structures during test execution contains no new methodological aspects.

5.1 TestDesign

The test design is a part of the EXAMProcess. It describes all elements needed for an automated generation of test cases in a runnable test program.


5.1.1 SpecificationLayer

The specification layer includes all elements needed for the description of the formal test specification and its structuring regarding reusability, inheritance, etc.


5.1.1.1 Terms [EXAM MM]

The table below contains all relevant terms for the specification layer.

Term:	Description:
AdministrativeCase	Special TestCase that is used for administrative tasks (e.g. initialization, deinitialization etc.)
EventClass	Represents an event in EXAM
EventClassAttribute	Attribute of an EventClass
EventClassInheritance	Inheritance between EventClasses. Every EventClass inherits from an abstract base class called <i>BaseEvent</i> .
EventClassOperation	Operation of an EventClass (e.g. send, create, check)
FormalTestIssue	<ul style="list-style-type: none"> - Precise, comprehensive description of a TestIssue - Formal representation of a TestIssue TestIssue – Element of the TestRequirements process step (see 5): Set of test topics that belong together with regard to content. Often also called test chapter.
FormalTestSpecification	<ul style="list-style-type: none"> - Aggregation of TestCaseSpecifications that belong together - Formalization of the InformalTestSpecification
FormalTestSpecificationClass	Concrete FormalTestSpecificationLanguage element to the encapsulate related FormalTestSpecificationExpressions
FormalTestSpecificationClassInheritance	Inheritance between FormalTestSpecificationClasses
FormalTestSpecificationExpression	Term defined in the FormalTestSpecificationLanguage that can be used within a TestCaseSpecification.
FormalTestSpecificationLanguage	Formal language to specify TestCases.
InterruptibleAction	Special TestAction. Its execution can be influenced by Events.

Concept paper	EXAM EXtended Automation Method	Department: I/EE-65	
		Page: 23 of 104	
		Revision date: 7/29/2009	

Term:	Description:
MappingClass	Blueprint for accessing objects on variables whose values are abstracted using a specific allocation table
MappingClassAttribute	Attribute of a MappingClass
MappingClassInheritance	Inheritance between MappingClasses
MappingClassOperation	Operation of a MappingClass (read and/or write access to MappingClassAttributes)
ParameterAttributeAssignment	Concrete assignment of a value to a ParameterClassAttribute
ParameterClass	Access class to test data resp. test parameters
ParameterClassAttribute	Attribute of a ParameterClass
ParameterClassInheritance	Inheritance between ParameterClasses
ParameterClassInstance	Instance of a ParameterClass to assign actual values to ParameterClassAttributes
ParameterClassOperation	Operation of a ParameterClass (read access to ParameterClassAttributes)
ParameterSet	Unit of test data which belongs together with regard to contents. Used to detail a TestFlow
ShortNameClass	The class provides a container for short names which serve as an alias for MappingClassAttributes or EventClassAttributes.
ShortNameClassAttribute	Attribute of a ShortNameClass
Template	Special TestActivity that serves as a template for concrete TestActivities. A derived TestActivity inherits the model elements from the Template.
TestAction	One and only one (1) action within a TestActivity
TestActivity	Description of sequential and parallel sequence of operations for testing a unit under test
TestActivityInheritance	Inheritance between TestActivities
TestCase	Description of exactly one (1) scope of testing in an abstract, formal and structured form. Variants of a common test flow are a separate TestCase each
TestCaseAttribute	Attribute of a TestCases
TestCaseSpecification	Formal specification of exactly one (1) full or partial TestCase
TestFlow	Abstract description of a sequence of operations for testing a unit under test
TestParameter	Unit of test data which belongs together with regard to contents. Used to detail exactly one (1) TestCase
TestParameterSetInheritance	Inheritance between ParameterSets

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	24 of 104	
		Revision date:	7/29/2009	

Term:	Description:
TestProcedure	Unit of TestFlows that belong together with regard to contents. Used to detail one and only one (1) TestCase
TestSequence	Sequence of operations for testing a unit under test
TestStep	One and only one (1) action within a TestSequence

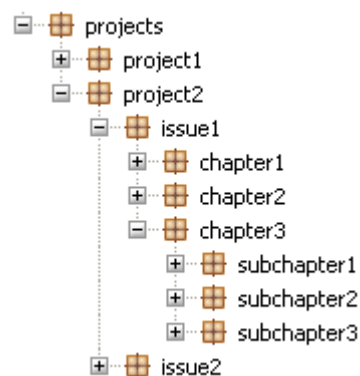
Relation between terms from the SpecificationLayer

5.1.1.3 Modeling

This chapter introduces the UML notation of elements from the SpecificationLayer in an EXAM project.

5.1.1.3.1 FormalTestIssue

A FormalTestIssue is modeled as a package in EXAM and is used for structured storage of TestCases or the FormalTestSpecification respectively. The FormalTestIssue term will generally be used in a broader scope. Any form of structuring in an EXAM model, e.g. for structuring libraries, is called FormalTestIssue. In addition to the package name, its textual description is also important in EXAM.



FormalTestIssues as packages

5.1.1.3.2 TestCases

A TestCase in EXAM is a use case marked with the «testCase» stereotype. The description field of the use cases contains a textual description of the TestCase containing information e.g. about its fundamental objectives, existing conditions, significant actions or the expected test results.

The formal description of a TestCase can be done using one sequence- or (XOR) activity diagram, which is assigned to the use case. The test designer may choose one or the other presentation form depending on the contents of the TestCase.

Tag Definitions can be used to attach any number of TestCaseAttributes to the *testCase* stereotype. These usually contain additional information about a TestCase. The list below describes the attributes that are defined by default in EXAM. All attributes are optional except for the *testCaseId*. A *testCaseId* must be provided by the test designer and it must be unique throughout the EXAM model.

testCaseId	Unambiguous labeling of a (obligatory)
testCaseState	Status of the TestCase (n/a, not yet specified, specified, reviewed, not yet implemented, implemented, productive, invalid)
duration	Expected duration of the test execution
version	Version number
implementationPriority	Priority with which this TestCase should be implemented/handled (low, middle, high)
riskEvaluation	Risk evaluation (latent, low, middle, high)
functionalRequirement	IDs of functional requirements that are assigned to this TestCase



TestCase as use case

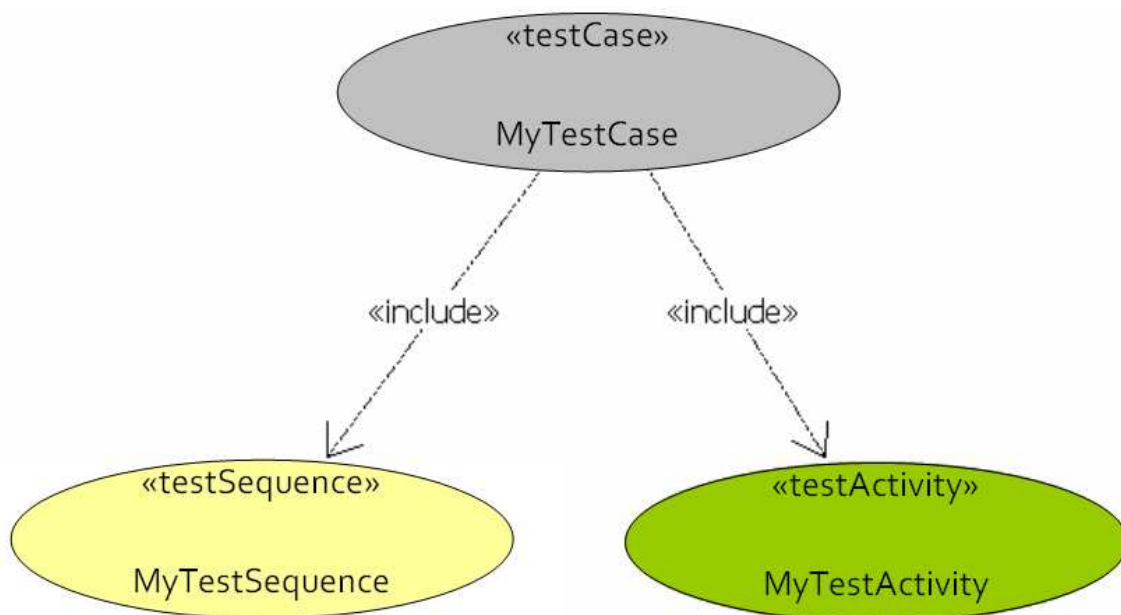
TestCases form the basis for modeling the TestExecution (see 5.1.4). When modeling the test structure, need may be to describe TestCases that are not relevant with regards to content but are inevitable for a successful test run. These special TestCases are called AdministrativeCases in EXAM. They can be used to e.g. initialize or deinitialize a device or platform. AdministrativeCases are handled just like regular TestCases, but they are explicitly marked with the «administrativeCase» stereotype.



Example for an AdministrativeCase

5.1.1.3.3 TestProcedure

Test cases usually have a structure. A widely used division is, for example, "precondition", "action" and "evaluation". To create such a division, EXAM defines include relationships between use cases.

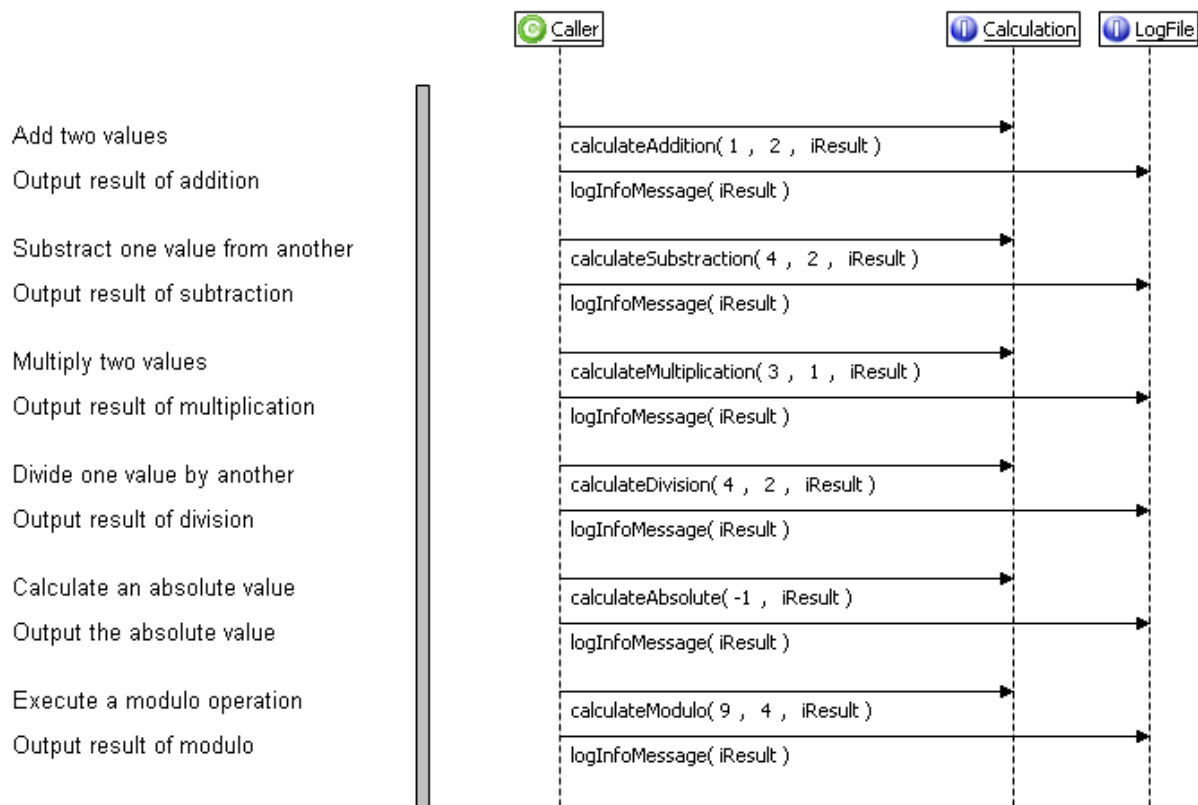


Structuring of a TestCase using include relations

The use case depicted in yellow is called TestSequence (see 5.1.1.3.3.1); the green one is a TestActivity (see 5.1.1.3.3.2). In the majority of cases, they do not represent a standalone TestCase but rather a structurally meaningful unit which then can be used in other TestCases as well. It is also possible to reuse TestCases as part of other TestCases. All TestFlows of a TestCase along with the TestCase itself form the TestProcedure.

5.1.1.3.3.1 TestSequences

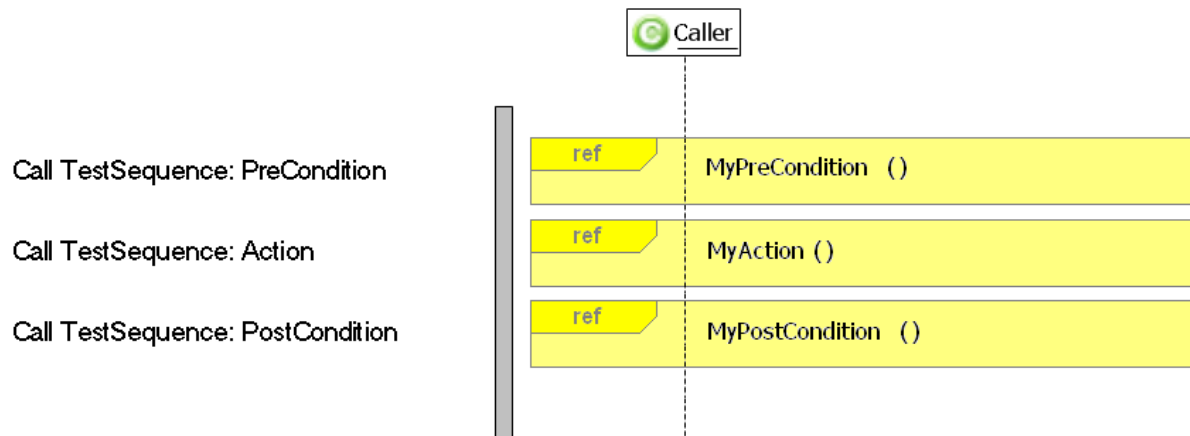
TestSequences are represented by use cases with the «testSequence» stereotype assigned. They are described by exactly one sequence diagram which is attached to that use case. The figure below shows an example of a simple EXAM TestSequence.



Example for a TestSequence

The time course within the TestSequence goes from top to bottom. *Calculation* and *LogFile* are class instances including their particular life lines (vertical, dashed line). The operation calls are carried out by sending messages from the caller class (see 6.10). The description of each step is shown on the left side of the system border (gray bar). It is comparable to comments in source code. Every sequence diagram has its own namespace.

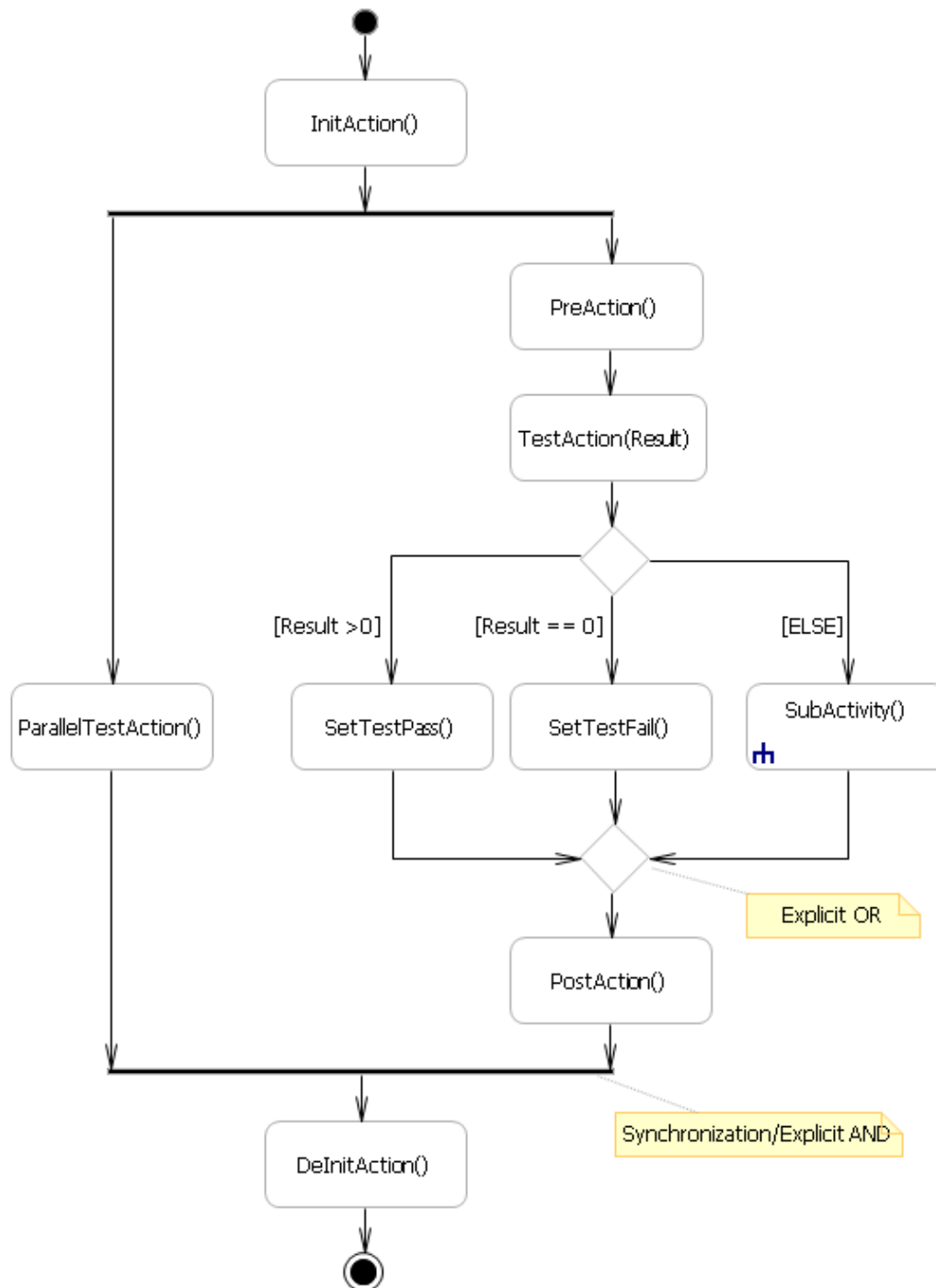
The specification of TestProcedures with use cases and include relationships, by definition, contains no information about the call order of TestSequences. In a sequence diagram, which is attached to a use case of the type *testCase* or *testSequence*, you have to explicitly specify the order of calls to TestSequences or TestActivities. Calls to TestFlows from within a sequence diagram are modeled using interaction frames of type *ref*.




Call of TestFlows from within a sequence diagram

5.1.1.3.3.2 TestActivities

TestActivities are represented by use cases with the «testActivity» stereotype assigned. They are described by exactly one activity diagram which is attached to that use case. The figure below shows an example of TestActivity with all description elements supported by EXAM.



Schematic TestActivity

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	32 of 104	
		Revision date:	7/29/2009	

● represents the entry point into the TestActivity. The area is left again at ●. The TestActivity can be left in more than one place. The white boxes are the actual actions. They include calls to class libraries, and thus are the counterpart to the messages in sequence diagrams.

The Sub_Activity() indicated in the figure, however, allows the integration of use cases and is, therefore, comparable to the interaction frames of type *ref* in a TestSequence.

TestActivities offer the possibility to parallelize the control flow. This is currently not allowed in EXAM TestSequences. The black bars symbolize the parallelization resp. synchronization point in the model (fork and join). The control flow will only continue if all tokens have arrived at the synchronization point.

The white diamond is called a decision. It allows to branch the control flow, i.e. the token follows the path of that branch, whose GuardCondition is met. GuardConditions are written into the square brackets adjacent to the outgoing edges. They are logical expressions that evaluate to a value equal (FALSE) or not equal (TRUE) to 0. If the GuardCondition of an edge evaluates to TRUE, the token continues to travel along this edge. If more than one GuardConditions have a TRUE value, the behavior is not deterministic. Therefore, EXAM provides the concept to index decision outputs (see 6.11.4). The [ELSE] GuardCondition provides an alternative path through the diagram, in case that none of the established GuardConditions evaluate to TRUE.

The diamond not only symbolizes the branch but also the join of the alternative paths through the TestActivity. TestActivities each have their own namespace, just like TestSequences.

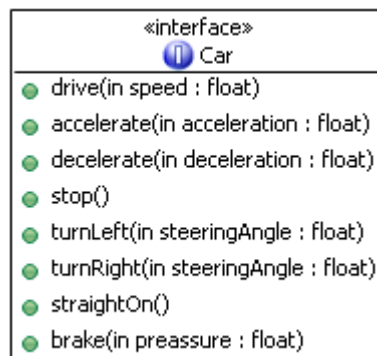
5.1.1.3.4 FormalTestSpecification

The visualization of the relationships between TestCases, TestSequences, TestActivities and ParameterSets (see 5.1.1.3.6) shall be made by the FormalTestSpecifications. These are use case diagrams with the «testSpecification» stereotype assigned. To adapt the level of detail of the diagram to fit desired application, it is expressly allowed to only describe a part of the entire TestCase.

5.1.1.3.5 FormalTestSpecificationLanguage

The FormalTestSpecificationLanguage is the vocabulary, based on which TestCases can be modeled. This language is designed to design TestCases without already implementing them. The FormalTestSpecificationLanguage is a level of abstraction in EXAM that enables you to model independently of a concrete platform or target system and hence allows for exchanging test cases.

The FormalTestSpecificationLanguage is based on interface classes (classes with the «Interface» stereotype assigned). Interfaces only contain operations, not attributes. The operations represent the vocabulary, which is available to test designers in EXAM models. They only provide the signature for the eventual implementation (see 5.1.2.3.1). Interfaces may inherit from each other. Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).

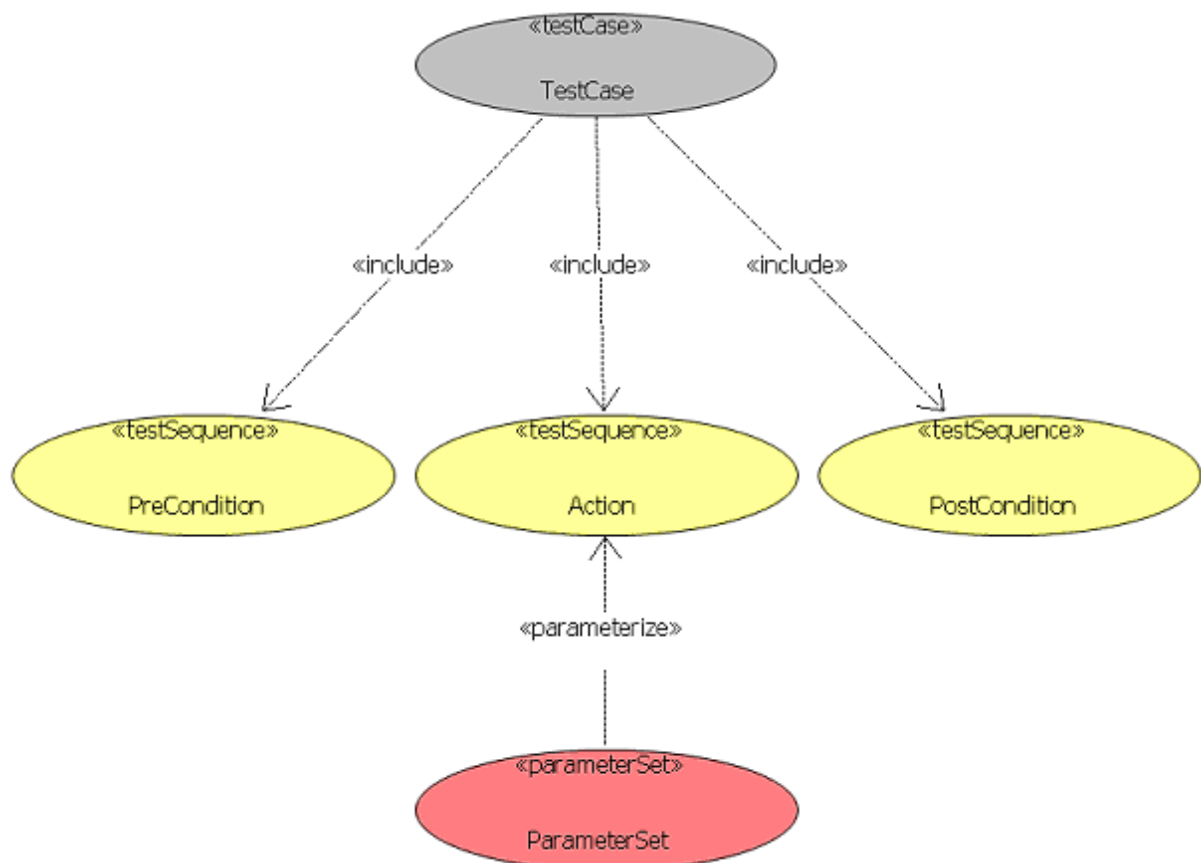


Example: Interface as part of the FormalTestSpecificationLanguage

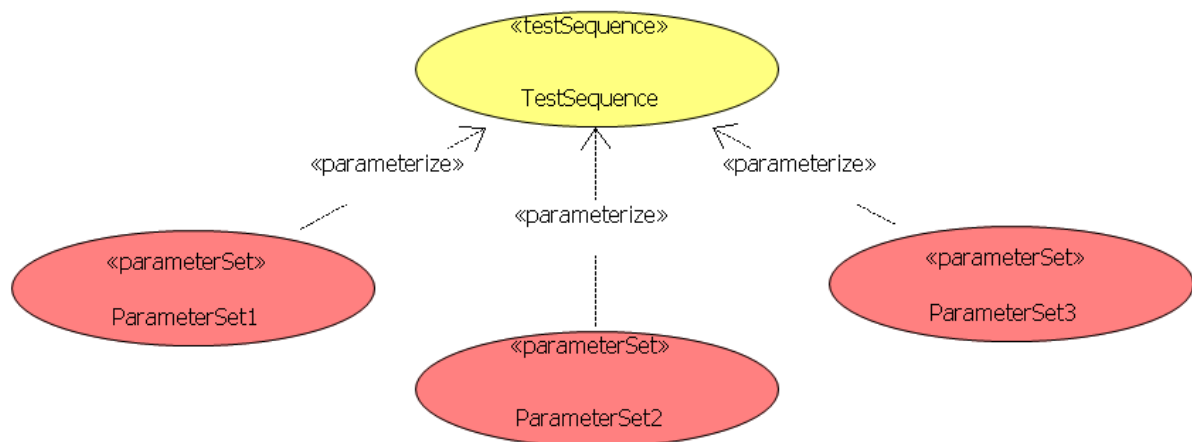
To ease working with large amounts of interfaces in the EXAM development environment, they can be grouped. By assigning the «interfaceGroup» stereotype with the related *name* tag to an interface, you can freely name interface groups.

5.1.1.3.6 ParameterSets

Another important basic concept of EXAM is the possibility to separate test data (parameters) and TestFlows. This is done using ParameterSets. ParameterSets are represented by a use case with the «*parameterSet*» stereotype assigned. They are connected to a TestFlow using the «*parameterize*» relation. You can associate any number of ParameterSets with a TestFlow. Within a TestCases, the parameterization must always unambiguous (see also 6.5.1).

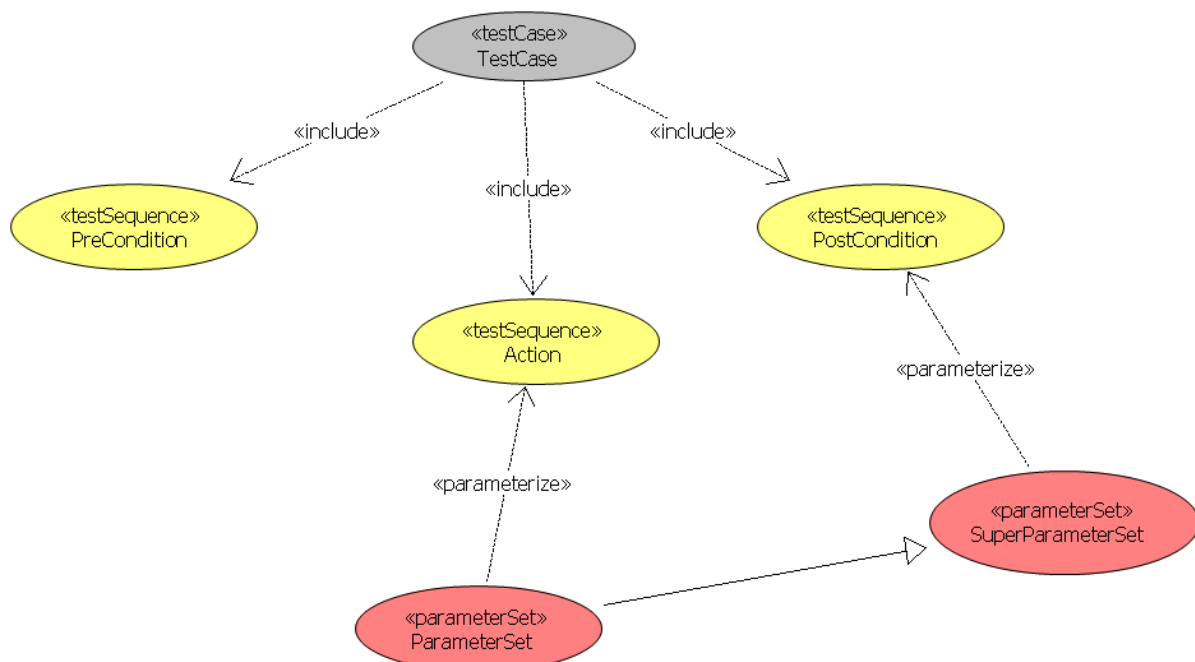


Assignment of a ParameterSet to a TestSequence within a TestCase



Assignment of multiple ParameterSets to a TestSequence

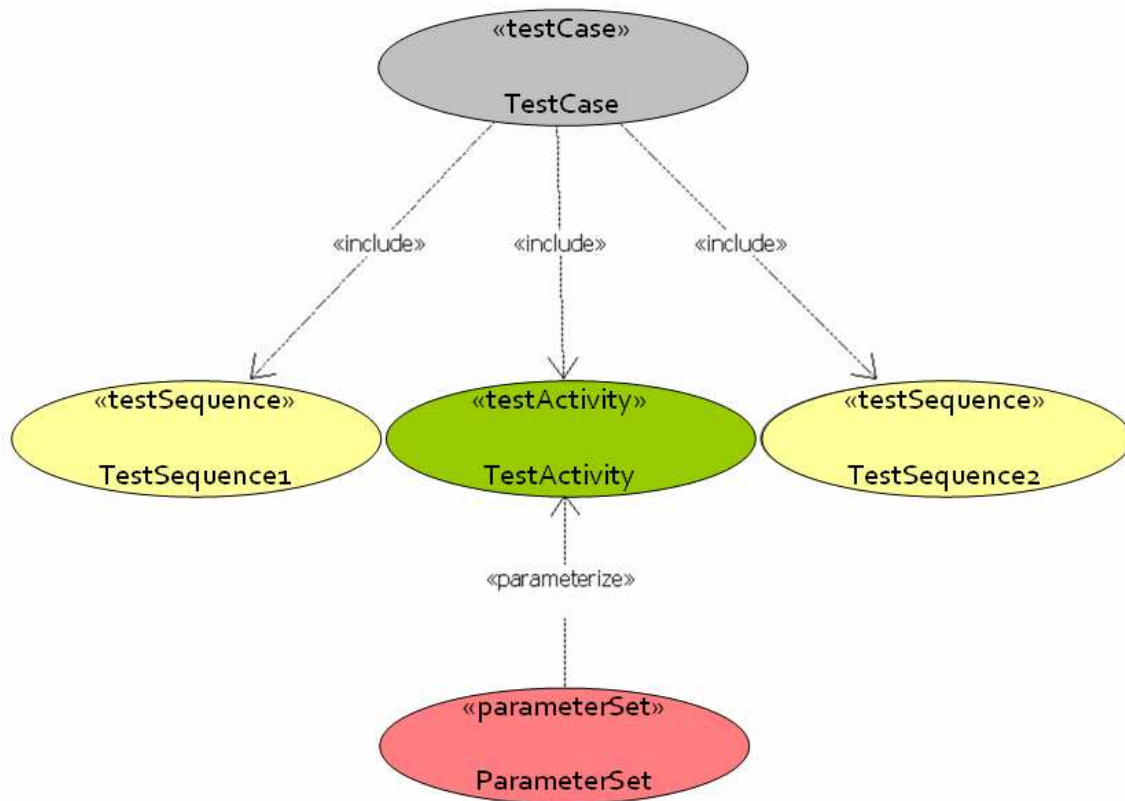
To improve reusability of ParameterSets it is possible to define inheritance hierarchies for ParameterSets in EXAM. This allows you to encapsulate and reuse parameter for multiple TestFlows. Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).




Inheritance between ParameterSets

The *Action* TestSequence from the figure above can use parameters from both *ParameterSet* and *SuperParameterSet* ParameterSets, while *PostCondition* can only use those from the *SuperParameterSet* ParameterSet.

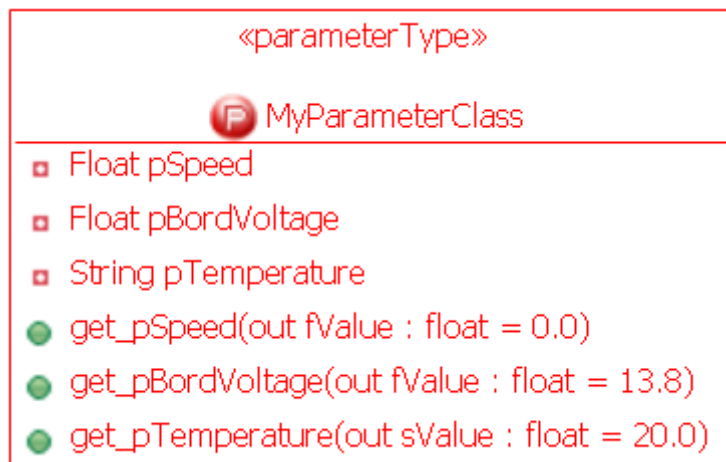
The parameterization works the same way for TestActivities.




Parameterization of a TestActivity

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	37 of 104	
		Revision date:	7/29/2009	

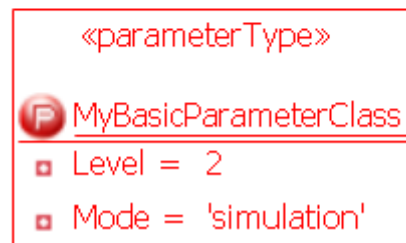
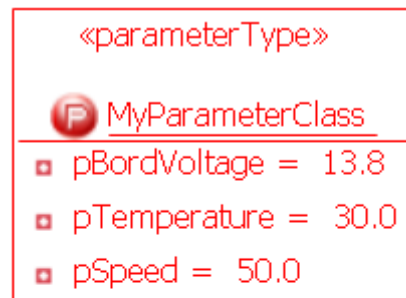
ParameterClasses form the foundation for defining ParameterSets. ParameterClasses are UML classes with the «*parameterType*» stereotype assigned. They possess a number of ParameterClassAttributes (the actual parameters) and one ParameterClassOperation (for read access) for each parameter. ParameterClassAttributes can have default values which are used when instantiating a ParameterClass. A parameter inheritance between classes is permissible. Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).



Example for a ParameterClass

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	38 of 104	
		Revision date:	7/29/2009	

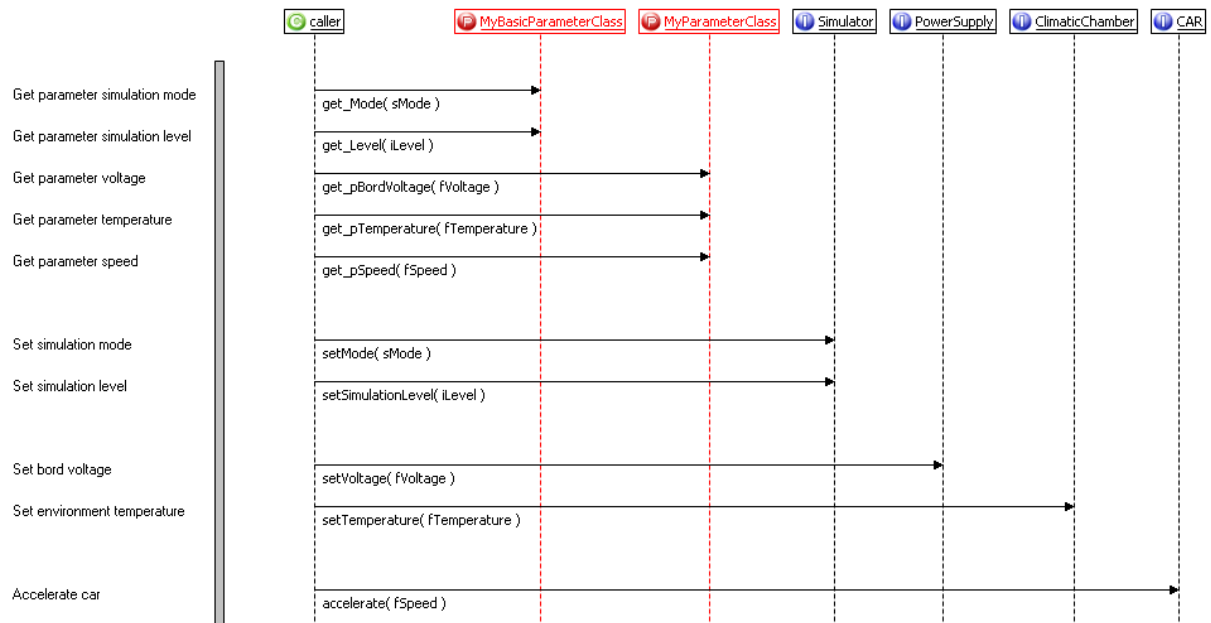
In order to define concrete values for a ParameterSet, an object diagram is attached to the use case. This enables the instantiation of the ParameterClass, so that the assignment of the values to the ParameterClassAttributes is possible. A ParameterClass must be instantiated only once on any particular object diagram.



ParameterClassInstances on an object diagram

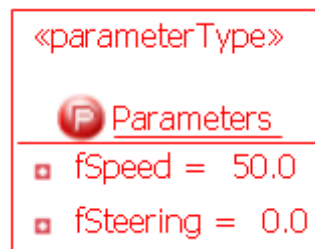
Based on the modeling shown here, the EXAM runtime environment takes care that the ParameterSet is accessible during execution of a TestFlow that makes use of it.

To gain access to the parameters, you explicitly need to make them available within a TestFlow using their get-operations. After that, they are stored in a local variable ready for further processing. Access to a ParameterClassInstance of a ParameterSet has to be carried out on the corresponding instance; or in other words: Identical instances of a ParameterClass have to exist in the ParameterSet and the TestFlow.

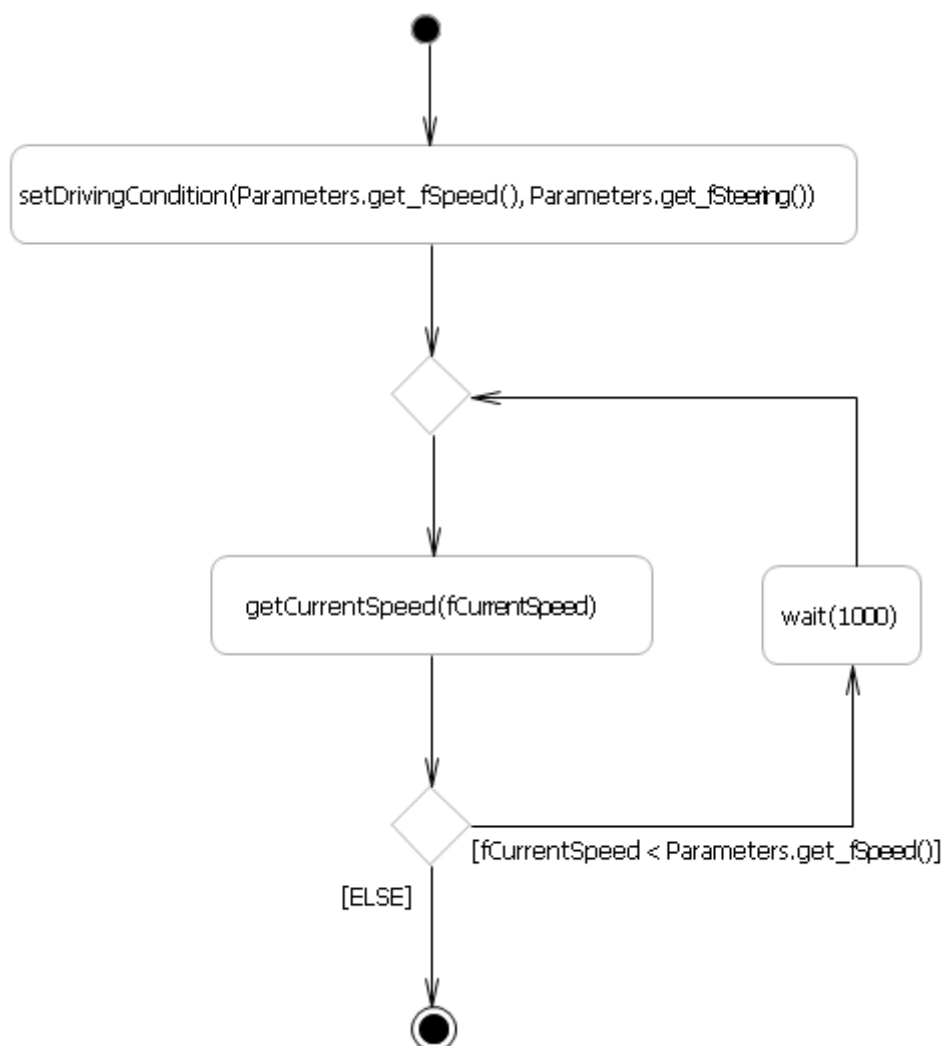


Usage of parameters within a TestSequence

Multiple instances of a ParameterClass on the same sequence diagram are not allowed.



Assignment of values in a ParameterClassInstance



Usage of parameters within a TestActivity

Please note:

In this scenario, the parameters are accessed by directly using the ParameterClassOperations in the call to a TestAction. The creation of out-parameters (see 6.8) can thus be omitted.

5.1.1.3.7 MappingClasses

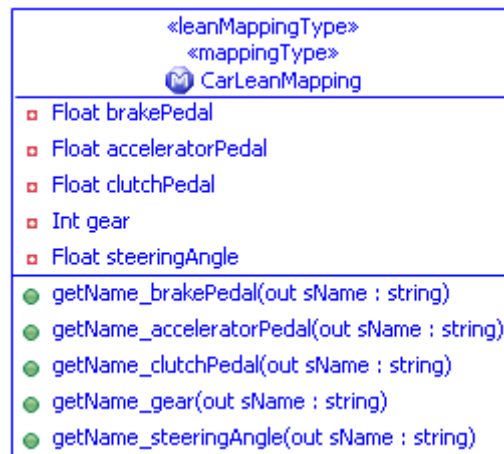
Mapping is another approach to decouple/abstract TestCases from the test system. It involves the assignment between a test automation- and a test system variable and is used to improve readability and exchangeability of TestCases.

To access the test system MappingClasses are used in EXAM. These are classes with the «*mappingType*» stereotype. The additional stereotype «*busMappingType*» indicates that the MappingClass contains signals to access bus systems (e.g. CAN, LIN, FlexRay, etc.). MappingClasses with the «*leanMappingType*» stereotype are treated specially by the development environment – the so-called EXAM Lean Mapping concept is used in these cases. For more information about the actual mapping concepts in EXAM, please refer to the EXAM documentation [DOC EXAM].

A MappingClassAttribute defines the name of a variable for use within the test automation system. MappingClassOperations are used for – depending on the underlying mapping mechanism – read (get-operations) and possibly write (set-operations) access to the variable or their model path/DataAccessPlatform (see 5.1.2.1) (getName-operations). Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).



MappingClass for 1-on-1 mapping concept



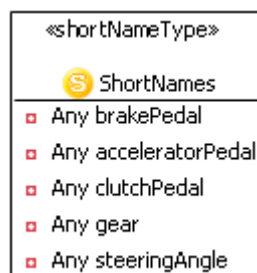
MappingClass for Lean Mapping concept

The return value of getName operations serve as a parameter for generic operations of access classes to test systems.

5.1.1.3.8 ShortNameClasses

All EXAM mapping concepts (see 5.1.1.3.7, 5.1.1.3.9, 5.1.2.3.3, 5.1.2.3.4) feature a two-stage abstraction approach. Within the FormalTestSpecification, it is possible to use ShortNameClasses as shorthand for Mapping- and EventClassAttributes.

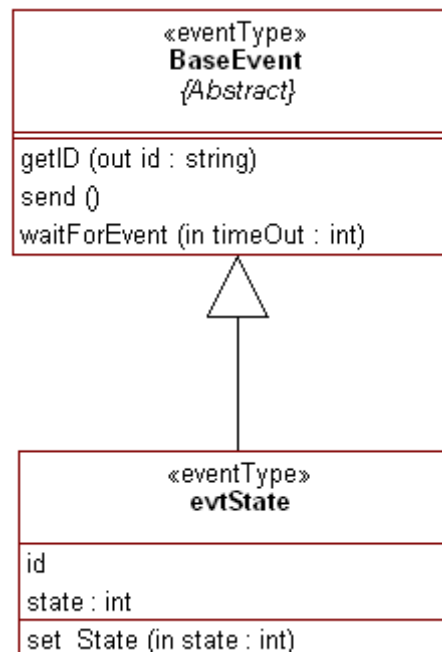
ShortNameAttributes are managed in ShortNameClasses. These are classes with the «shortNameType» stereotype. They have to be unambiguous within a Variable- or EventMapping. ShortNameClasses contain only attributes, not operations. Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).



Example for a ShortNameClass

5.1.1.3.9 EventClasses

EXAM offers the possibility to include events from target systems into the actual test design. Events are represented by classes with the «eventType» stereotype assigned. They must inherit from an abstract base class, *BaseEvent*. Multiple inheritance is not allowed. The figure below shows an example of an event class.

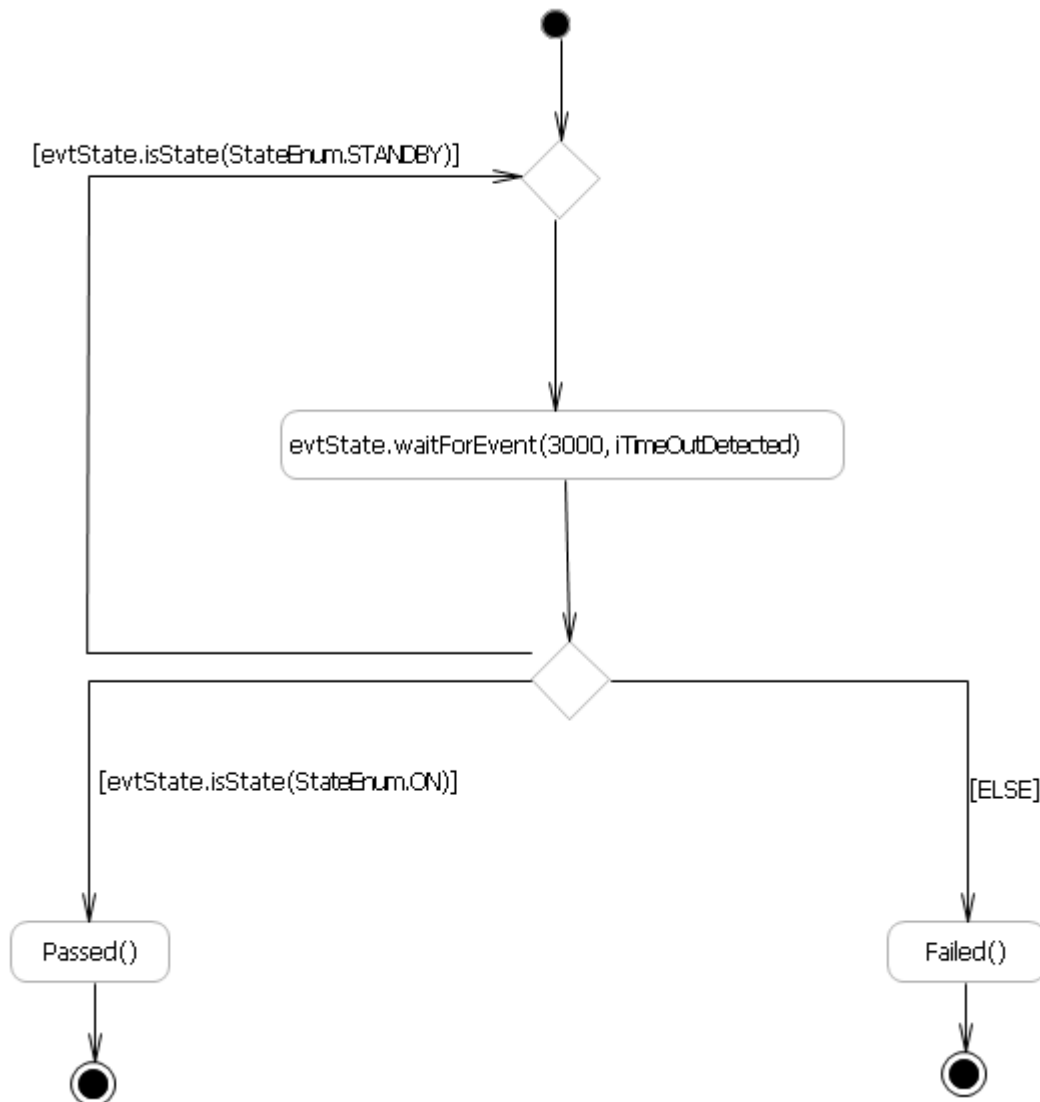


EventClass in EXAM

Concrete EventClasses have an own pool of values in form of EventClassAttributes and associated EventClassOperations. Usually, *set*-operations are used for filling the attributes. The *is*-operations can be used to evaluate attribute values.

The *getID* operation of the *BaseEvent* class provides a unique identification of a particular event within an EventMapping (see 5.1.2.3.4) or the event handling mechanism, respectively. To trigger events on a simulation platform, the *send()* operation can be used. *waitForEvent* offers a simple way to wait for events within a test process.

The figure below shows an example on how to use events in TestActivities.



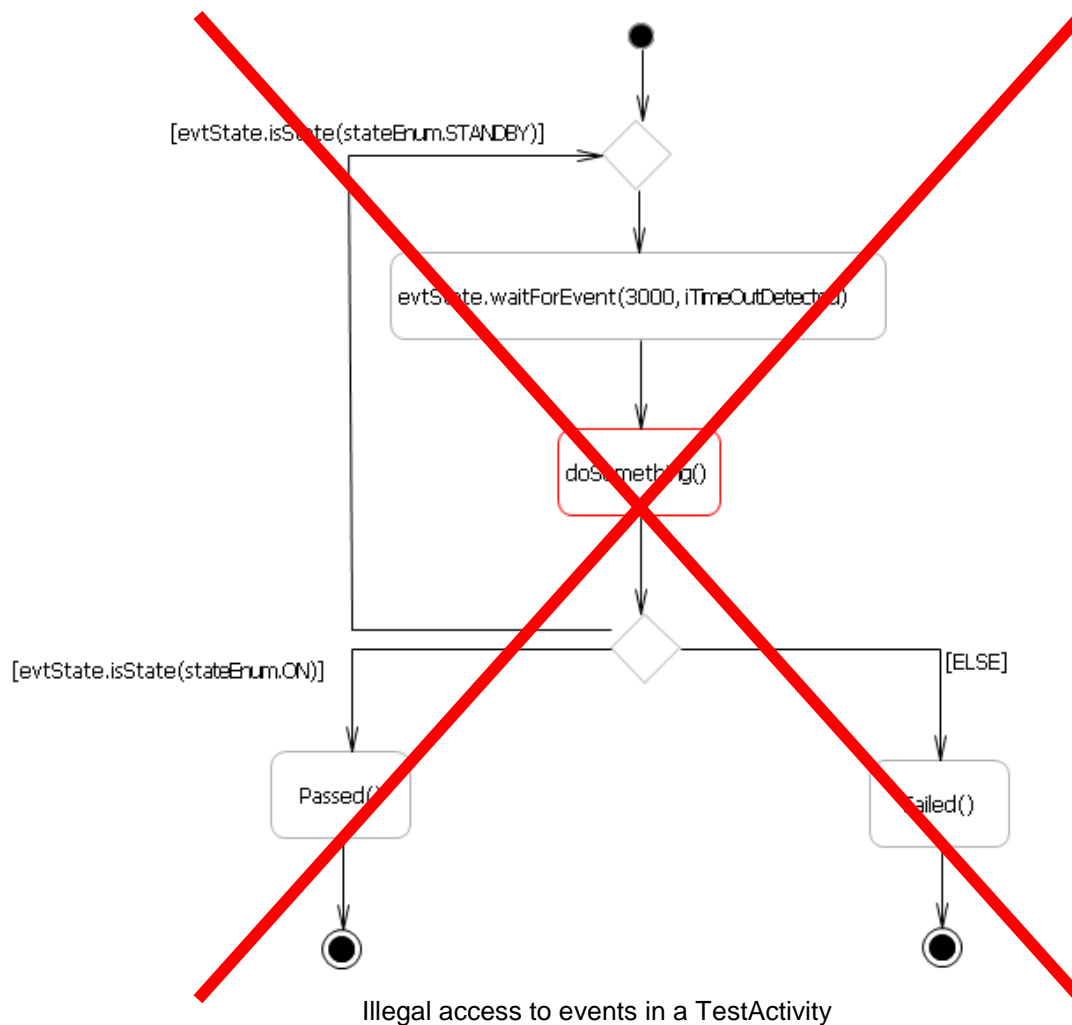
Events in TestActivities

Important Information:

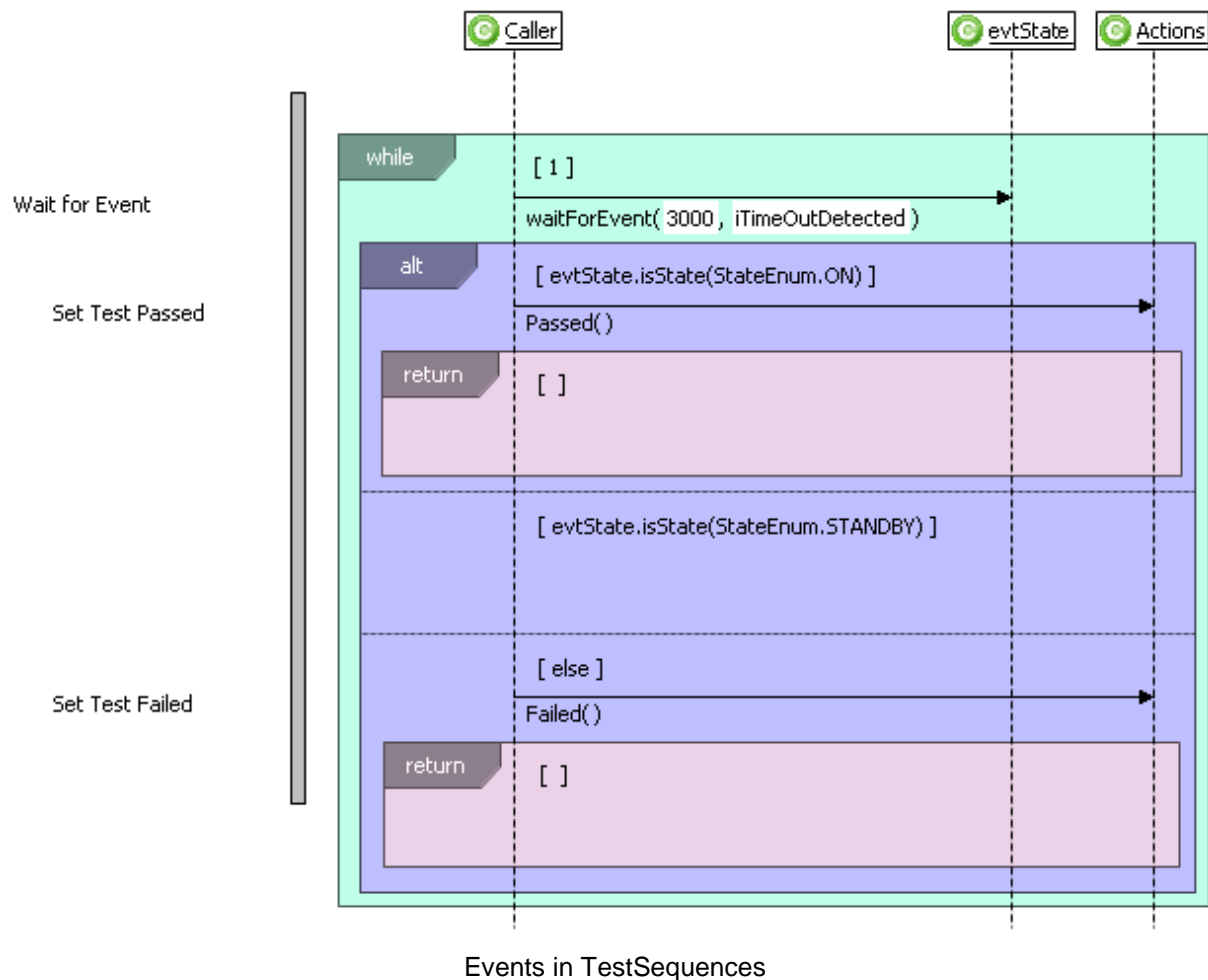
EXAM only allows the evaluation of EventClasses within GuardConditions directly after the *waitForEvent* operation, i.e. there must be no action between the GuardCondition and the evaluation. This holds true for the InterruptibleActions topic (see 6.11.6).

The reason for this restriction is the temporal validity of events. In EXAM, events can only exist in the period between their occurrence and their immediate evaluation.

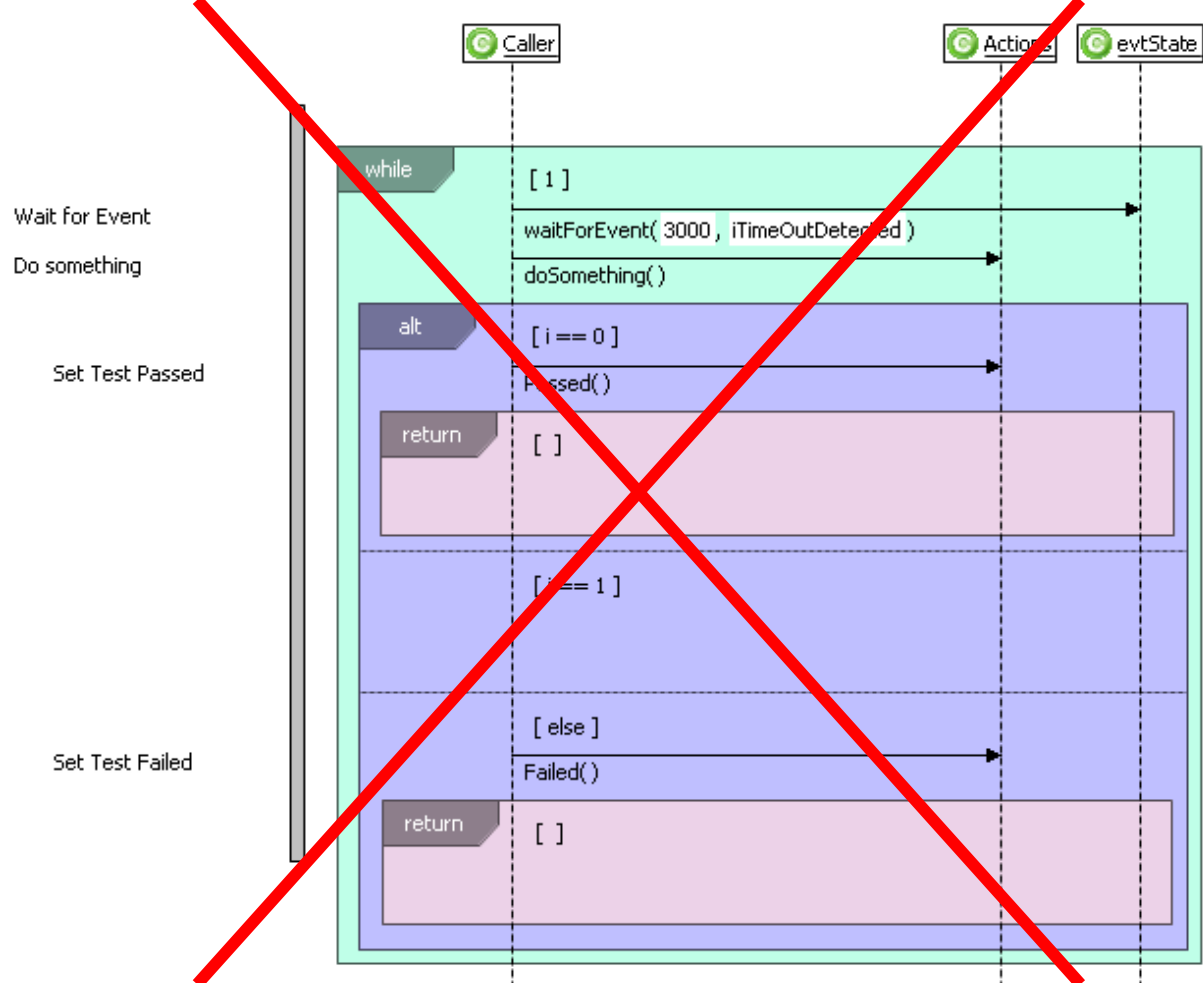
The figure below shows an example for an illegal evaluation of events.



These facts apply to sequence diagrams as well.



There must be no operation calls between the *waitForEvent* message and the evaluation of that event.



Illegal access to events in a TestSequence

5.1.2 ImplementationLayer

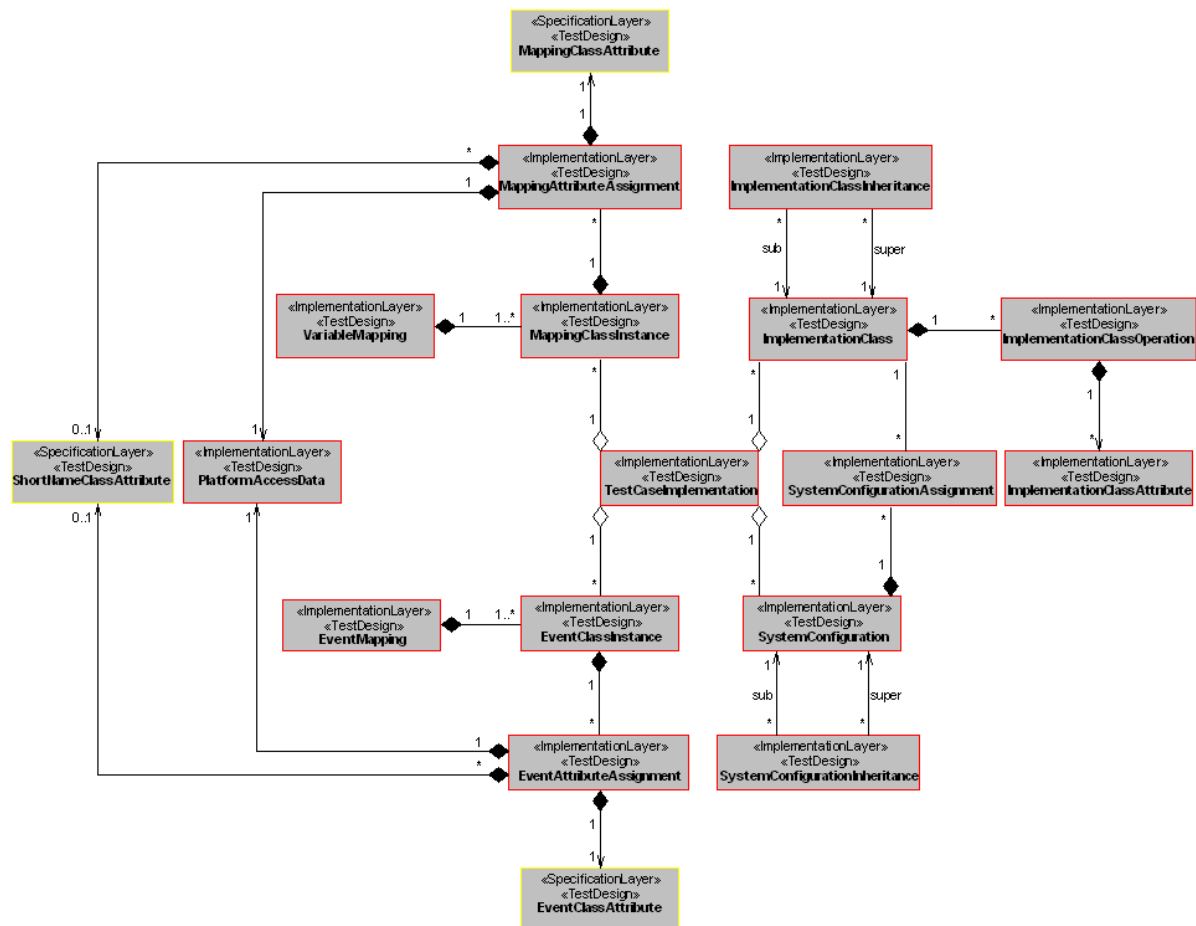
The ImplementationLayer contains all model elements that are required to implement the SpecificationLayer.

5.1.2.1 Terms [EXAM MM]

The table below contains all relevant terms for the implementation layer.

Term:	Description:
EventAttributeAssignment	Concrete assignment of values and ShortNameClassAttributes to an EventClassAttribute
EventClassInstance	Instance of an EventClass to assign concrete values to EventClassAttributes
EventMapping	Container for EventClassInstances
ImplementationClass	Container for operations that belong together with regard to content to implement a FormalTestSpecificationClass
ImplementationClassAttribute	Attribute of an ImplementationClass
ImplementationClassInheritance	Inheritance between ImplementationClasses
ImplementationClassOperation	Operation of an ImplementationClass
MappingAttributeAssignment	Concrete assignment of values and ShortNameClassAttributes to an MappingClassAttribute
MappingClassInstance	Instance of an MappingClass to assign concrete values to MappingClassAttributes
PlatformAccessData	Information that is required for physical access to a simulation platform, e.g. to access a path in a Simulink model.
SystemConfiguration	Definition which ImplementationClasses to use for which FormalTestSpecificationClasses. Required for executing tests on a test system.
SystemConfigurationAssignment	Assignment of exactly one ImplementationClass to exactly one FormalTestSpecificationClass
SystemConfigurationInheritance	Inheritance between SystemConfigurations
TestCasImplementation	Generic term for all elements needed for the implementation of exactly one TestCase
VariableMapping	Container for MappingClassInstances

5.1.2.2 Relation between terms form the EXAM meta model [EXAM MM]



Relation between terms from the ImplementationLayer

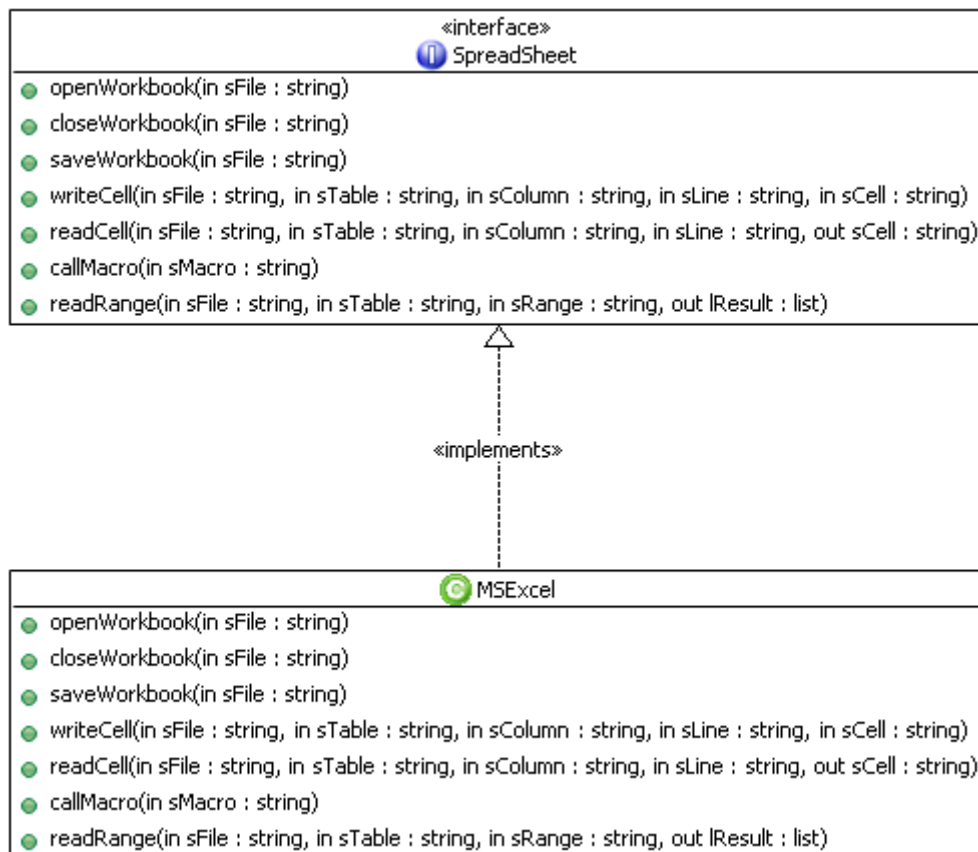
5.1.2.3 Modeling

This chapter introduces the UML notation of elements from the ImplementationLayer in an EXAM project.


5.1.2.3.1 ImplementationClasses

Classes that realize interfaces of the FormalTestSpecificationLanguage are called ImplementationClasses. The operations of these ImplementationClasses can be (even within a class), implemented either with a sequence- or activity diagram or with code fragments from the target programming language. Sequence- and Activity diagrams must be subordinate to the particular operation in the EXAM model.

An implementation is connected to its interface with an «implements» dependency relationship. The signature ImplementationClassOperations must exactly match those defined in the FormalTestSpecificationExpressions. ImplementationClasses may contain additional operations. Furthermore, the actual default values (see 6.8) are permitted to differ from those defined in the FormalTestSpecificationClass. Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).

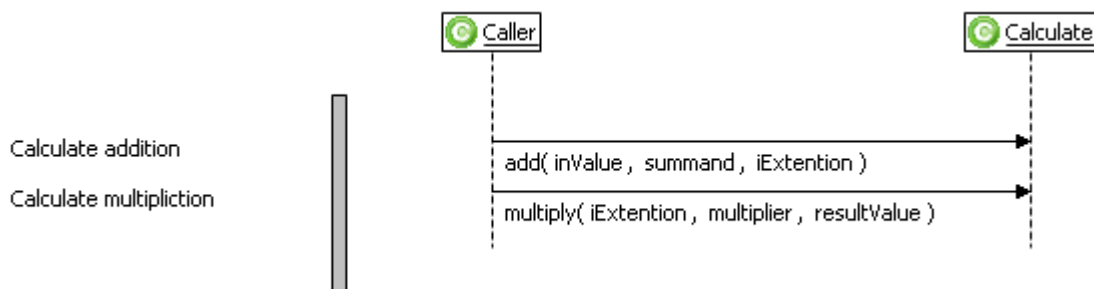


The *MSExcel* class implements the *SpreadSheet* interface

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	51 of 104	
		Revision date:	7/29/2009	

Calculate
<ul style="list-style-type: none"> add(in value1 : int, in value2 : int, out result : int) multiply(in value1 : int, in value2 : int, out result : int) addAndMultiply(in inValue : int, in summand : int, in multiplier : int, out resultValue : int)

Example for an ImplementationClass



Implementation of the *addAndMultiply* operation of class *Calculate* as sequence diagram

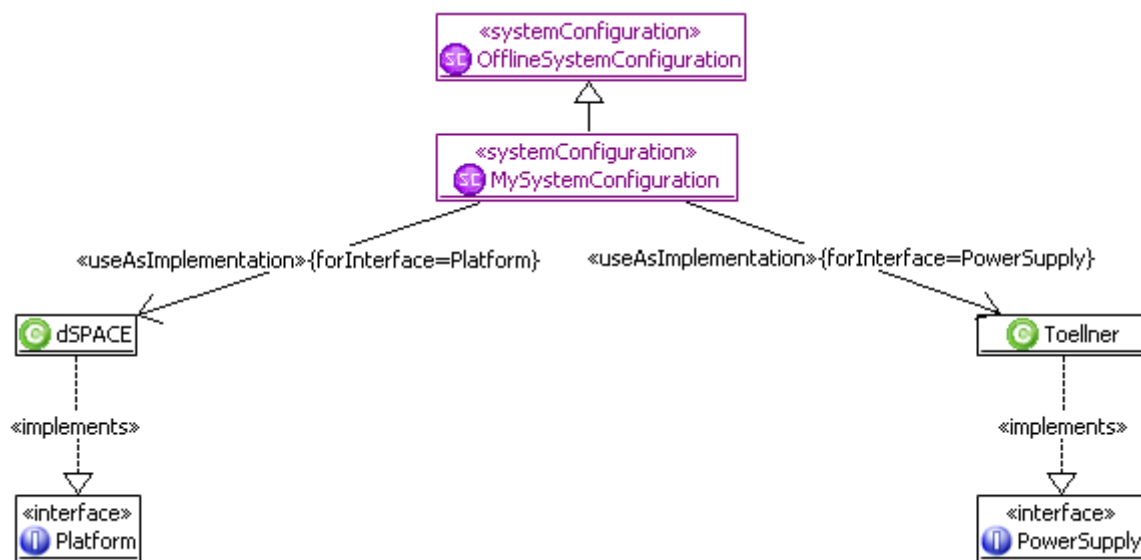
Important information:

ImplementationClasses be used in Test Cases as well as in FormalTestSpecificationClasses (interfaces). However, a direct use of ImplementationClasses would statically bind the particular implementation to the TestCase. This approach would eliminate one layer of abstraction – and thus one of EXAM's strengths. Generally, you should only use ImplementationClasses directly if indirection using interfaces does not make sense in your particular context and you do not expect it to become useful ever.

5.1.2.3.2 SystemConfigurations

In a typical EXAM model, there are a number of interfaces, each of which can be realized by one or more ImplementationClasses. A set of assignments between exactly one Implementation- and FormalTestSpecificationClass is called System Configuration.

A SystemConfiguration in EXAM is represented by class with the «systemConfiguration» stereotype assigned. It comprises all classes that shall be used as implementation within this configuration. The assignment is done via a directed 1:1 association with the «useAsImplementation» stereotype. The stereotype contains a *forInterface* TagDefinition of type Reference, which allows for creating the connection between the association and the FormalTestSpecificationClass. This is necessary because ImplementationClasses within a SystemConfiguration can implement multiple interfaces. System-Configurations in EXAM may inherit from one another. Multiple inheritance is not allowed.



SystemConfiguration

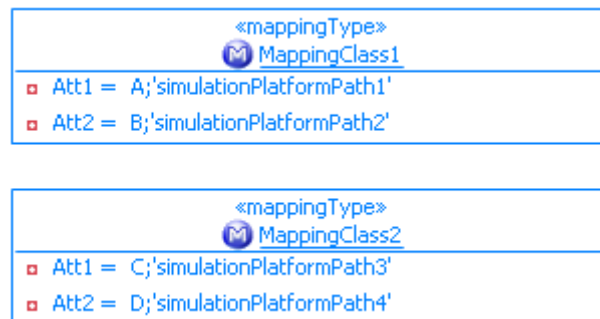
The SystemConfiguration *MySystemConfiguration* inherits all SystemConfigurationItems of class *OfflineSystemConfiguration*. In addition, the *dSPACE* class is defined as an implementation for the *Platform* interface and the *Toellner* class for the *PowerSupply* Interface.

5.1.2.3.3 VariableMappings

Similar to the System Configuration (see 5.1.2.3.2), you need to specify the mapping between MappingClassAttributes, ShortName-ClassAttributes and associated simulation platform paths when implementing a FormalTestSpecification. This is done by so-called VariableMappings.

Variable Mappings are classes with the «*variableMapping*» stereotype assigned. Each VariableMapping class has exactly one object diagram attached. All MappingClasses that belong to a particular VariableMapping are instantiated on its object diagram.

A reference to a ShortNameAttribute and a PlatformAccessData to access the simulation platform variable is assigned to each MappingClassAttribute of every MappingClassInstance on the object diagram. The two parts of those assignments are strings, separated by a semicolon. Inheritance between VariableMappingClasses is not allowed.



Object diagram of a VariableMappingClass in EXAM

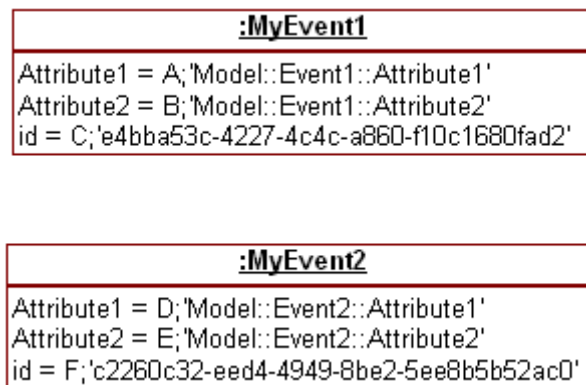
The figure above shows two MappingClassInstances on the object diagram of a VariableMapping-Class. The MappingClassAttributes Att1 and Att2 each have a ShortNameAttribute (A, B, C and D) and a simulation path assigned.

5.1.2.3.4 EventMappings

When implementing a `FormalTestSpecification`, you need to map `EventClasses` to their clearly defined, unique identifier on the target simulation platform. In EXAM, the `EventMapping` concept is used to achieve that.

`EventMappings` are classes of type «*eventMapping*». Each `EventMappingClass` has exactly one object diagram attached. All `EventClasses` that belong to a particular `EventMapping` are instantiated on its object diagram.

A reference to a `ShortNameAttribute` and a `PlatformAccessData` to access the simulation platform is assigned to each `EventClassAttributes` of every `EventClass` on the object diagram. Note that this also applies for the *id* attribute that has been inherited from the *BaseEvent* class (see 5.1.1.3.9). The two parts of those assignments are strings, separated by a semicolon. Inheritance between `EventMappingClasses` is not allowed.



Object diagram of an `EventMappingClass` in EXAM

The figure above shows two `EventClassInstances` on the object diagram of an `EventMappingClass`. The `EventClassAttributes` `Attribute1`, `Attribute2` and `id` each have a `ShortNameAttribute` (*A*, *B*, *C*, *D*, *E* and *F*) and an identification string assigned.

5.1.3 CompositionLayer

The composition layer comprises all model elements to parameterize EXAM TestCases.

5.1.3.1 Terms [EXAM MM]

The table below contains all relevant terms for the composition layer.

Term:	Description:
Composition	<u>Alternative 1:</u> Assignment of exactly one ParameterSets to exactly one TestFlow <u>Alternative 2:</u> Statement that no ParameterSet is required for one particular TestFlow
TestCaseComposition	Set of all Compositions that are needed for exactly one TestCase

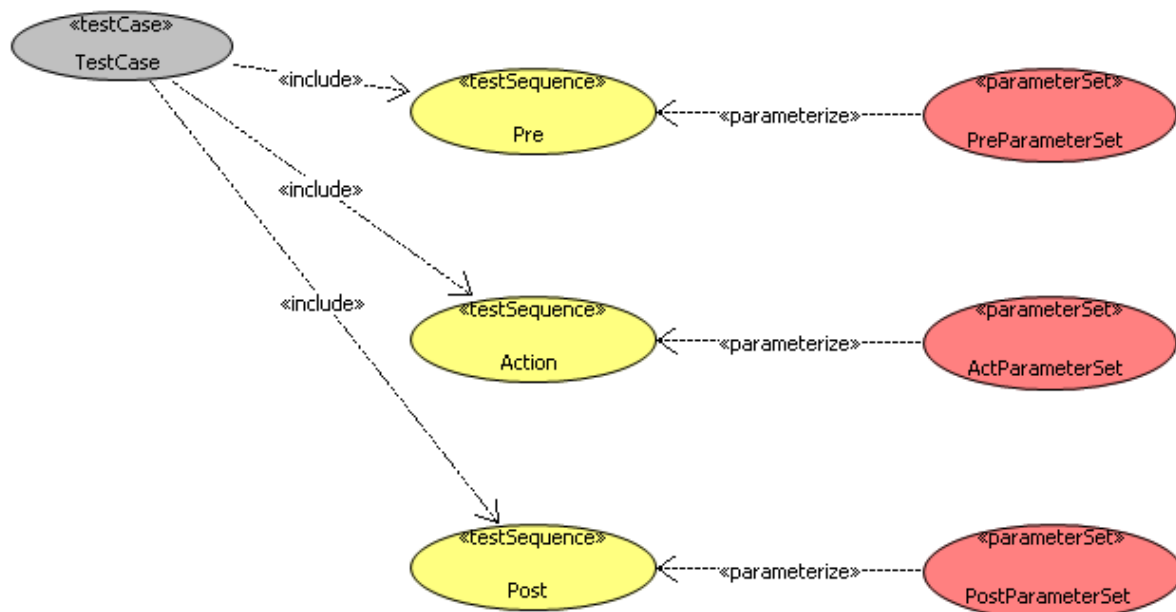
5.1.3.2 Relation between terms form the EXAM meta model [EXAM MM]




Relation between terms from the CompositionLayer

5.1.3.3 Modeling

The TestCaseComposition determines which specific parameterization for a TestCase is used. This is done with use case diagrams with the «testCaseComposition» stereotype assigned. A diagram includes the TestCase and all of its TestFlows as well as their associated ParameterSets. The parameterization must be unambiguous. Super ParameterSets will not be shown in the diagram. The TestSpecification diagram and the TestCaseComposition diagram can be one and the same, but then it has to have both stereotypes («testSpecification» and «testCaseComposition») assigned. The TestSpecification diagram is identical to the TestCaseComposition diagram or contains a subset thereof.



TestCaseComposition diagram

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	57 of 104	
		Revision date:	7/29/2009	

5.1.4 ExecutionLayer

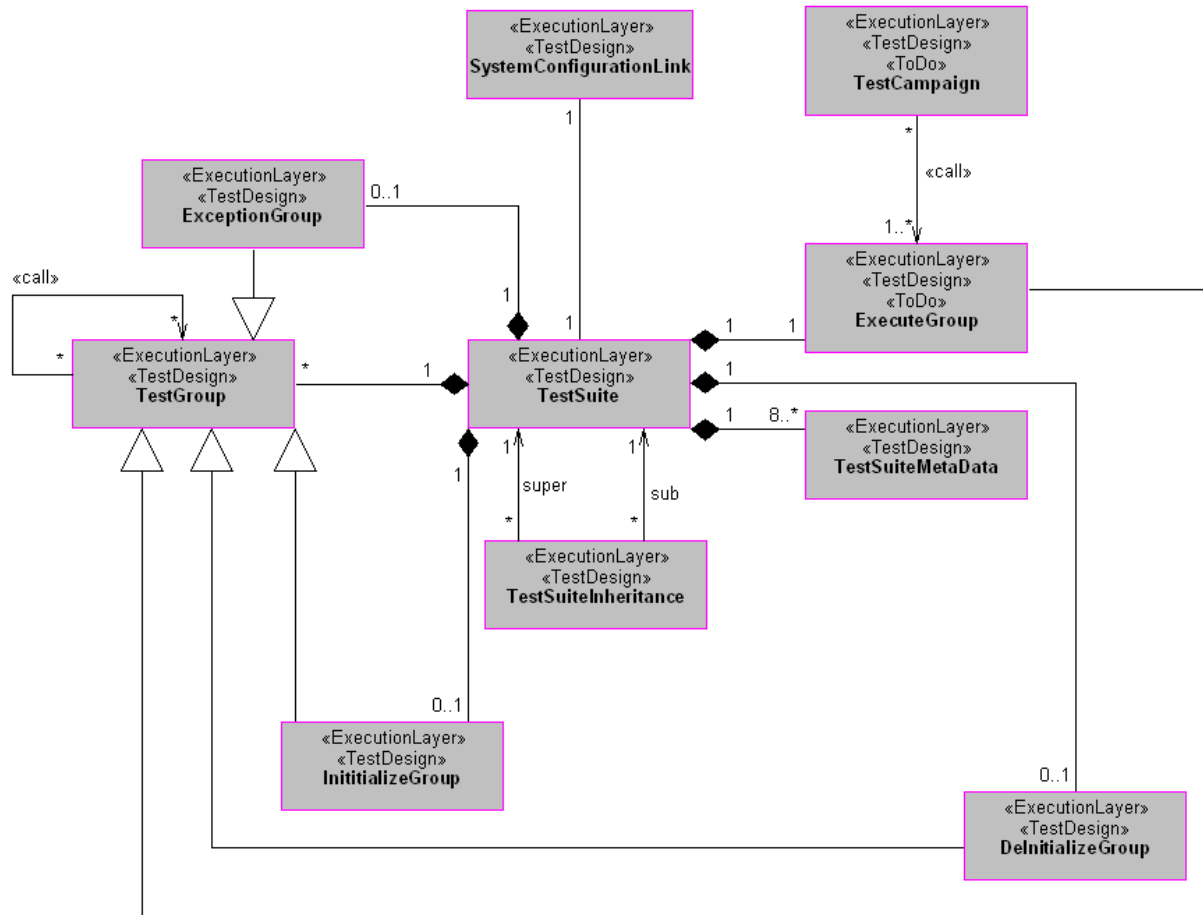
The execution layer comprises all model elements to define execution units.

5.1.4.1 Terms [EXAM MM]


The table below contains all relevant terms for the execution layer.

Term:	Description:
DeInitializeGroup	Group within a TestSuite to combine deinitialization tasks
ExceptionGroup	Group within a TestSuite to combine error handling tasks
ExecuteGroup	Special group that is used in TestCampaigns to execute TestSuites
InitializeGroup	Group within a TestSuite to combine initialization tasks
SystemConfigurationLink	Static assignment of one or more SystemConfigurations to a TestSuite
TestCampaign	Execution unit. Defines the call order of contained TestSuites
TestGroup	Group within a TestSuite to structure TestCases
TestSuite	Grouping of TestGroups and TestCases that belong to a certain test topic
TestSuiteInheritance	Inheritance between TestSuites
TestSuiteMetaData	Attribute of a TestSuite to define meta data

5.1.4.2 Relation between terms form the EXAM meta model [EXAM MM]



Relation between terms from the ExecutionLayer

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	59 of 104	
		Revision date:	7/29/2009	

5.1.4.3 Modeling

This chapter introduces the UML notation of elements from the execution layer in EXAM projects.

5.1.4.3.1 TestSuite

A Test Suite is a self-contained unit that defines the order in which TestCases are executed. It must not make any assumptions about other TestSuites that may or may not run before or after this one or even depend on them to execute required tasks. Instead, a TestSuite has to all required preconditions by itself. For each TestSuite a separate EXAM ResultSet (see 5.2.1) is generated. In addition, a TestSuite represents an enclosed area of responsibility within a test project.

TestSuites are classes with the «testSuite» stereotype assigned. They have a number of attributes which can be used to define meta data. The operations of a TestSuite are called TestGroups. They are used to structure the TestSuite and to specify the execution order of the TestCases. Therefore, each group has a sequence diagram attached. Calling TestCases from the TestSuite is done using the *ref* interaction frame (see 5.1.1.3.3). Test cases must only be called from TestGroups by definition. Each TestGroup has a *runs* parameter that determines how often it should be executed. If this parameter is not specified, the default value is used instead. TestGroups may contain any number and combination of TestGroups themselves. TestSuites may inherit from each other. Multiple inheritance is permissible according to the EXAM inheritance rules (see 6.5).

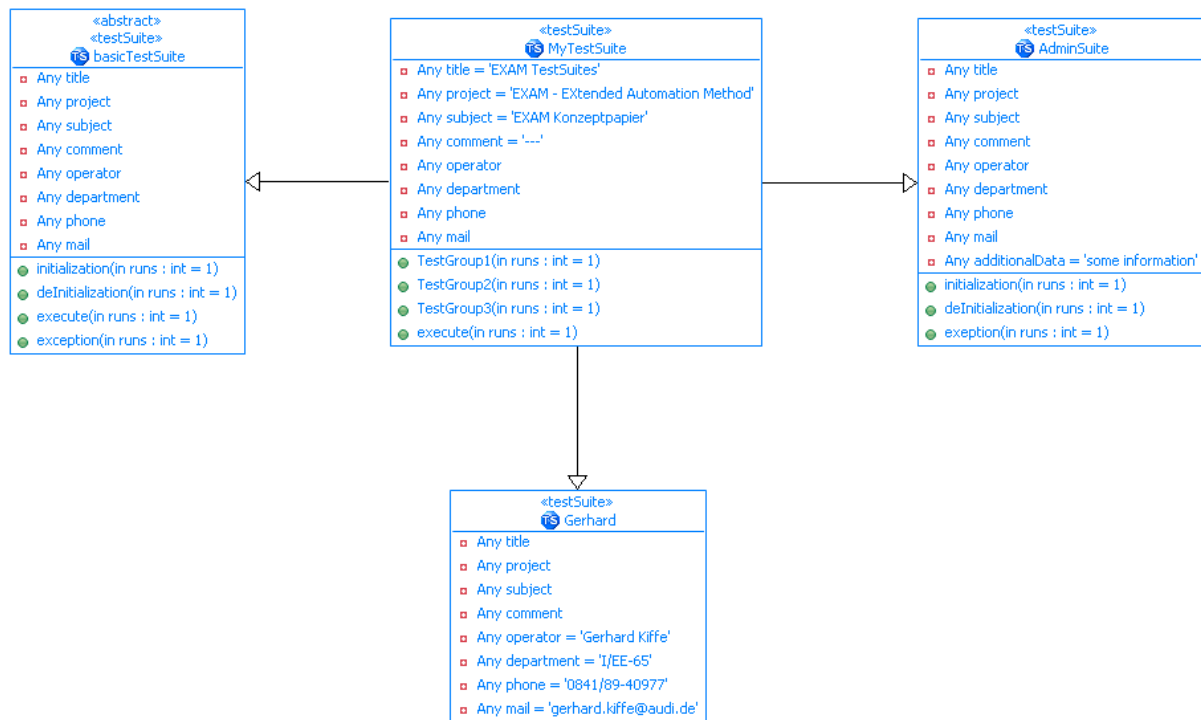
For a TestSuite to be executable in EXAM, it must contain a minimum level of functionality and information. Therefore, every TestSuite is derived from the *BasicTestSuite* class. This abstract class contains both the required minimum meta data, as well as necessary structural units/TestGroups for execution. TestSuites that do not contain those required attributes and operations are permitted in EXAM; however, they can not be executed.

The *initialization* and *deInitialization* TestGroups are used to initialize and deinitialize the test system. The *initialization* TestGroup is always executed before the execution of the actual TestCases, even if you restart a TestSuite and continue test execution with any of the TestCases from within the TestSuite. The *deInitialization* TestGroup is executed in any case, even if the execution of the TestSuite is terminated by request of a TestCase or the test user.

The *exception* TestGroup defines an exceptional call sequence, which is executed in response to an internal exception during a test run.

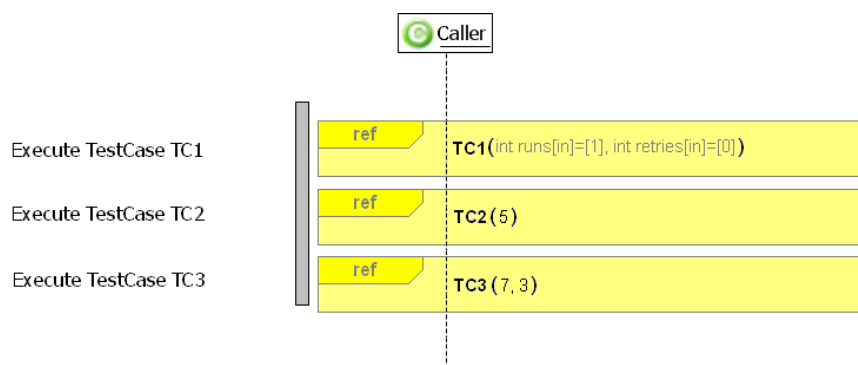
The *execute* TestGroup will be called by a TestCampaign to execute the TestSuite. It represents the entry point to the TestSuite. TestCases must not be called directly from the *execute* TestGroup.

TestSuites can be linked to SystemConfigurations. This is done by the *systemConfigurationLink* tagged value, which references one or more SystemConfiguration classes. The EXAM run time system will automatically change the SystemConfiguration upon start of a TestSuite. If no System-Configuration is given for a particular TestSuite, the default ones are used. These default configurations are selected when you start a test run.



Example TestSuites

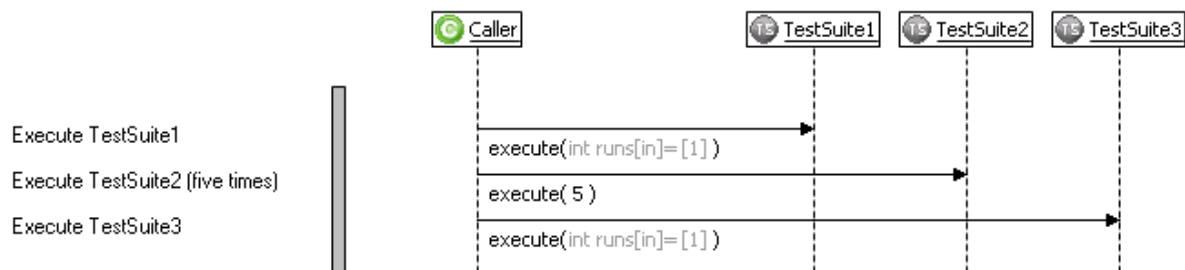
Besides the *runs* argument, you can also pass the *retries* parameter to a call to a TestCase within a TestGroup. It specifies how often the run time system should retry to run the TestCase in case an exception occurred.



Calling TestCases from a TestGroup

5.1.4.3.2 TestCampaign

TestCampaigns configure the call sequence of TestSuites. They are modeled using sequence diagrams with the «testCampaign» stereotypes assigned. TestSuites are executed by sending a message to their *execute* TestGroup/operation. TestCampaigns can be used to control how many ResultSets (see 5.2.1) are created during a test run.



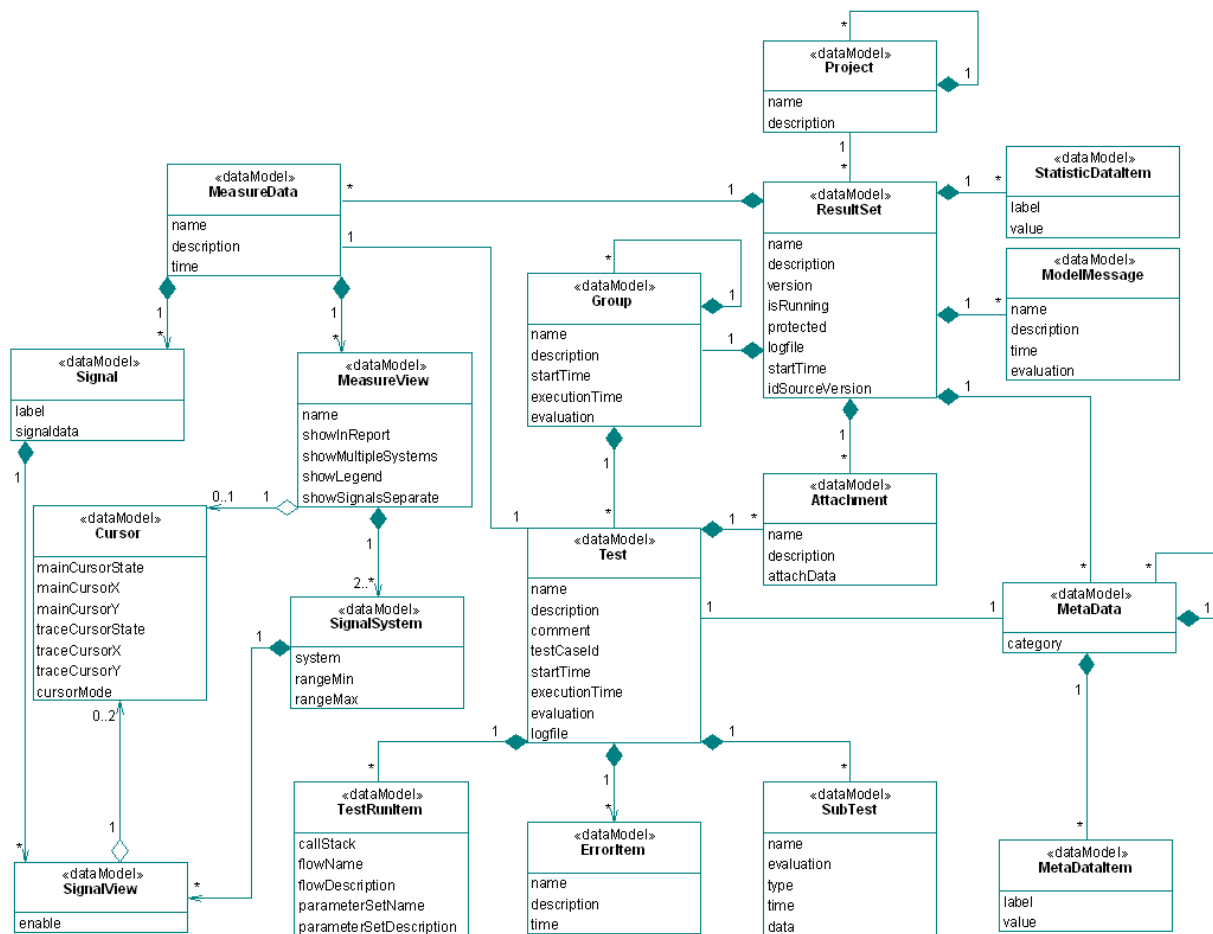
Example TestCampaign

5.2 TestOutput


For the **TestOutput** process step it is important to know, which information can or must be identified, defined and evaluated during a test run. This chapter introduces the different classes of result data that exist in EXAM.

5.2.1 Data model for result data

The figure below shows the relations between the types of result data.




Data model: EXAM result data


Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	63 of 104	
		Revision date:	7/29/2009	

5.2.2 Definition of entities and attributes


Entity	Description
Project	A Project is a container for all ResultSets that belong to a FormalTestIssue. A project can also contain other Projects. They are represented by packages in the UML model.
name	Name
description	Description
ResultSet	A ResultSet aggregates information gathered during execution of a TestSuite to an administrative unit.
name	Name
description	Description
version	Version number
isRunning	This attribute indicates whether a ResultSet is currently being filled with result data, i.e. if the associated TestSuite or the processing of result data is still running.
protected	Attribute to control write access to the data of a ResultSet
logfile	Contents of the master logfile
startTime	Creation time
idSourceVersion	Reference to the source version of the ResultSet
MetaData	MetaData is a structural unit that contains any number of MetaDataItems or other MetaData. Used to structure meta data.
category	Name of the MetaData structural unit

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	64 of 104	
		Revision date:	7/29/2009	


Entity	Description
MetaDatum	MetaDataItems are carriers of general information about a test or a ResultSet. You can, for example, further describe the unit under test, the test system or the test controller. There can be various sources for meta-information.
label	Descriptive information about a meta date, e.g. chassis number
value	Actual meta information, e.g. WAUZZZ 44 ZKN
StatisticDatum	StatisticDataItems are carriers of statistical information on a test run or ResultSet. Examples include the number of test cases, the number of tests with a specific valuation, the frequency of occurrence of a defined event during a test run etc.
label	Descriptive information about a statistic date, e.g. the number of tests with valuation "pass"
value	Actual statistic information, e.g. 256
ModelMessage	Container for information from asynchronous messages that are not explicitly assigned to a specific TestCase. They can for example be generated by a simulation platform.
....name	Name
....description	Description
time	Date and time of arrival
evaluation	Valuation
Attachment	Attachments are arbitrary files that can be attached to a ResultSet or test.
name	Name
description	Description
attachData	Actual (file) content

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	65 of 104	
		Revision date:	7/29/2009	

Entity	Description
Group	A Group is used to structure the ResultSet. It corresponds to a TestGroup from the UML model.
name	Name
description	Description
startTime	Execution start time
executionTime	Duration of the Groups execution
evaluation	Overall valuation
Test	A Test contains all information that has been generated during the execution of a TestCase.
name	Name
description	Description
comment	Arbitrary comment
testCaselId	ID of a Test. Should be unique within the originating model.
startTime	Starting time
executionTime	Duration
evaluation	Overall valuation
logfile	Contents of the associated logfile
TestRunItem	A TestRunItem contains information about the TestSequences, TestActivities and ParameterSets that belong to a TestCase.
callStack	Attribute to determine the call hierarchy of the TestCase
flowName	Name of an executed TestFlow
flowDescription	Description of an executed TestFlow
parameterSetName	Name of a used ParameterSet
parameterSetDescription	Description of a used ParameterSet

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	66 of 104	
		Revision date:	7/29/2009	

Entity	Description
ErrorItem	An ErrorItem contains information about an error that occurred during the execution of a TestCase
name	Name
description	Description
time	Point in time when the error occurred
SubTest	A SubTest represents detailed information on a Test. It contributes to the overall valuation of that Test.
name	Name
evaluation	Valuation
type	The type of SubTest defines the structure in which the subtest data is stored. Possible types are, for example, comment, fault memory entry, etc.
time	Entry date
data	The actual SubTest data
MeasureData	A set of raw data e.g. from a real time measurement
name	Name
description	Description
time	Save date
MeasureView	A view on a MeasureData
name	Name
showInReport	Flag to define whether a MeasureView should be included in a generated report document.
showMultipleSystems	Flag to define whether the recorded signals should be displayed using multiple Y axes.
showLegend	Flag to define whether the signal legend should be displayed.
showSignalSeparate	Flag to define whether each of the recorded signals should be displayed in a separate coordinate system.
Signal	Channel within a MeasureData
label	Name
signaldata	Recorded signal data

Concept paper	EXAM EXtended Automation Method	Department: I/EE-65	
		Page: 67 of 104	
		Revision date: 7/29/2009	

Entity	Description
SignalView	A view on a recorded signal
enable	Flag to activate and deactivate a signal in the SignalView
SignalSystem	An axis within the coordinate system
system	Type of the SignalSystem (e.g. X axis or Y axis)
rangeMin	Starting value of a display range of an axis
rangeMax	End value of a display range of an axis
Cursor	A pair of cursors (main- and traceCursor)
mainCursorState	Flag to activate and deactivate the mainCursor
mainCursorX	X position of the mainCursor
mainCursorY	Y position of the mainCursor
traceCursorState	Flag to activate and deactivate the traceCursor
traceCursorX	X position of the traceCursor
traceCursorY	Y position of the traceCursor
cursorMode	Flag to define the cursor mode. Can be one of FREE – Free position of the Cursor LOCK – Cursor can only be positioned on available measured value (one from the MeasureData)

5.2.3 Test valuation in EXAM

The result types described below are available to valuate test results.

Result type	Description
pass	The SUT passed the test.
fail	The SUT failed the test.
open	It is not possible to valuate the test because for example an error occurred or rated values are missing.
info	The result has a purely informative nature.

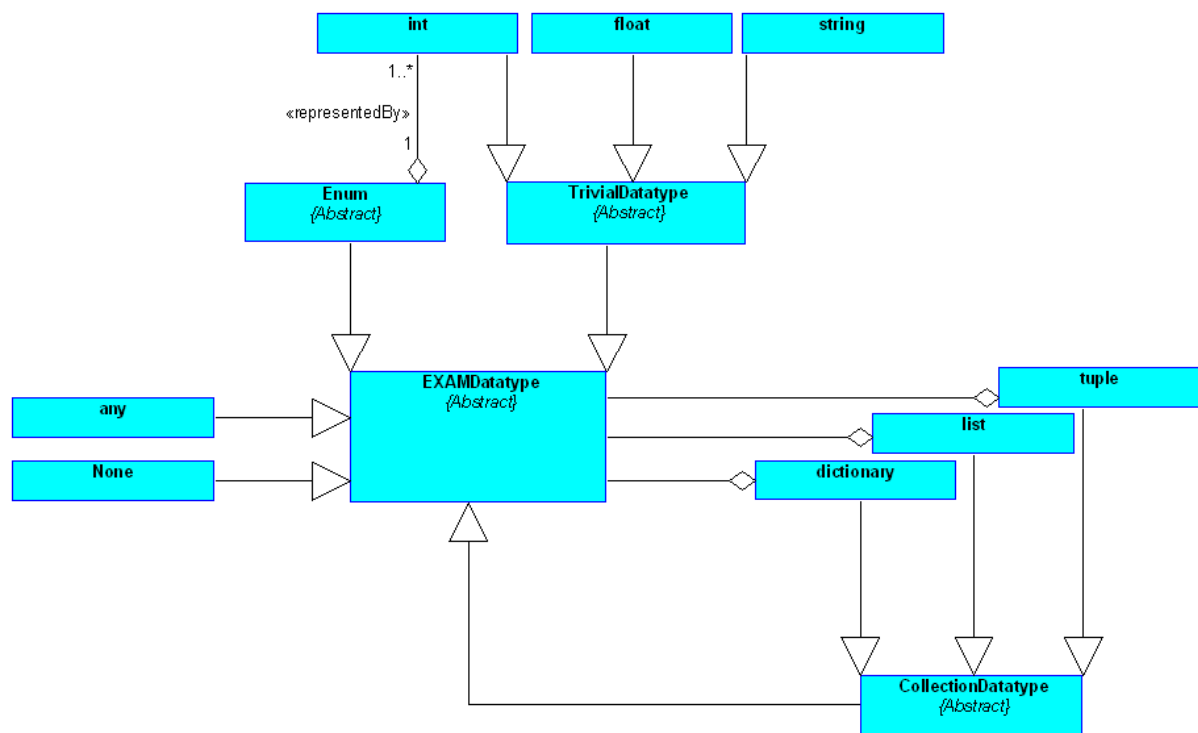
6 EXAM conventions

This chapter describes syntactic rules that apply for the creation of EXAM test models.

6.1 EXAM data types

This section specifies the EXAM data types. Basically, EXAM adapts the type concept of the Python programming language [PYTHON]. Specific restrictions apply.


Within an EXAM UML model, a number of atomic and structured data types can be used. The figure below provides an overview of the accepted data types in EXAM.



EXAM data types

Trivial data types:

Data type:	Description:
float	Floating point number
int	Integers
string	Character strings

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	69 of 104	
		Revision date:	7/29/2009	

Collection data types:


Data type:	Description:
tuple	Unchangeable vectors of object references, numerically indexed
list	Changeable vectors of object references, numerically indexed
dictionary	Changeable, indexed set of key-value pairs; accessed using keys

Miscellaneous data types:

Data type:	Description:
Any	Any of the EXAM data types
None	Placeholders for variables that actually have no value

All EXAM data types follow certain syntactical rules, so that their type can be inferred from their notation:

- **float:** Sequence of digits, which contains a point. Scientific notation (using “e”) is also possible.
Examples: 3.14 10. .001 1e100 3.14e-10 0e0
- **int:** Sequence of digits, an optional minus sign in front of it denotes negative numbers. Prefix 0x indicates a hexadecimal, 0 an octal number. A minus sign is only allowed for decimal numbers.
Examples: 1234 -987 0x2af 010
- **string:** Character strings enclosed in single or double quotes. The escape sequences listed below are recognized and treated specially.
 - \\ Backslash (\)
 - \' Single quote (')
 - \“ Double quotes (“)
 - \a ASCII Bell (Beep)
 - \n ASCII Linefeed (LF)
 - \t ASCII horizontal tab (TAB)*Example:* 'This is a single “string“. \n \'New line\'. \a'
- **None:** Object “None“
- **tuple:** Comma-separated list of constants. Enclosed in round brackets.
Example: ('a string', 1234, 3.21, ('a nested tuple', 9))
- **list:** Comma-separated list of variables. Enclosed in square brackets.
Example: ['n', 1234, 0.3234e21, ('a nested tuple', 9)]
- **dictionary:** Comma-separated list of key-value pairs enclosed in curly brackets. Key and value are separated by ":". Keys may only have an atomic data type. Values may have any basic data type. Duplicate keys will not lead to an error. The value of the rightmost key will override all others.
Example: {'have': 'haben', 'get': 'bekommen', 1234: ('a tuple'), 0.3234e21: {987: 'a nested dictionary'}, 'mSwKL30': 'DATA_EXCHANGE/Fehler_CAN/s_kl_30_zv.Value'}

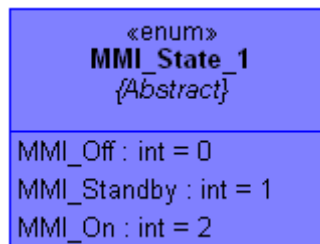
Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	70 of 104	
		Revision date:	7/29/2009	

Enums

EXAM supports untyped enumerations using symbolic names for numeric values. Enumeration types are modeled in EXAM as abstract classes with the «enum» stereotype assigned. The attributes of the class determine the values and their symbolic representation.

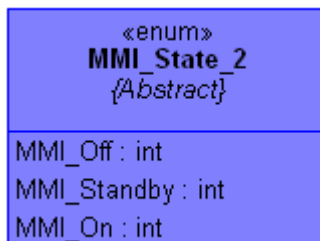
The «enum» classes are globally available within an EXAM model and can be used as any other data type. The usage of symbols is done by referencing the attribute name.

The assignment of values can be done implicitly or explicitly. The figure below shows an example of an explicit assignment.



Explicit definition of an Enum in EXAM

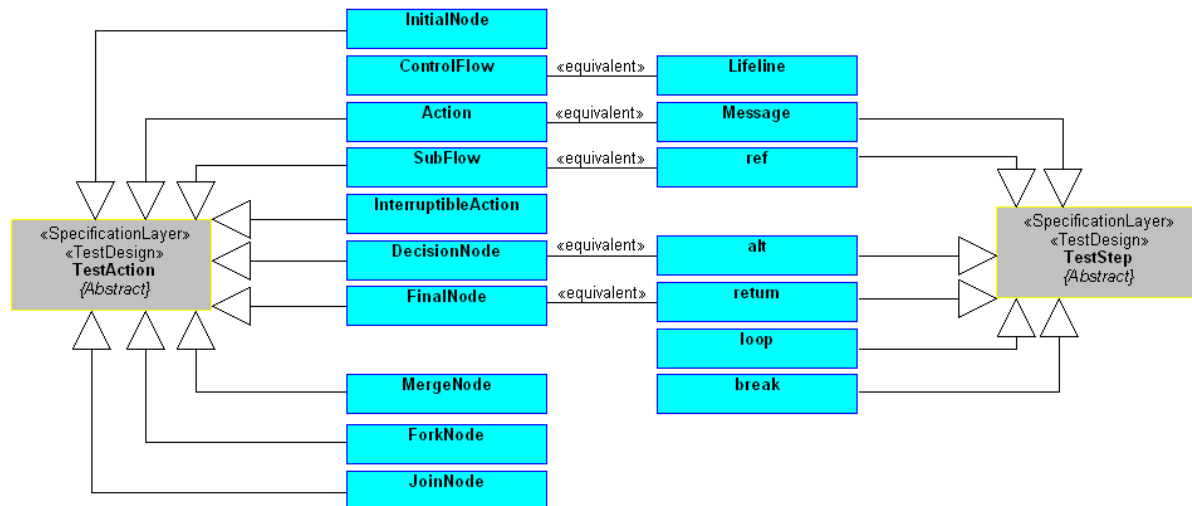
If no explicit values are given, they are assigned implicitly based on the order of the attributes within the class. The figure below shows an example of an implicit assignment. The values will be identical to the ones in the first example.



Implicit definition of an Enum in EXAM

6.2 Control flow elements in EXAM

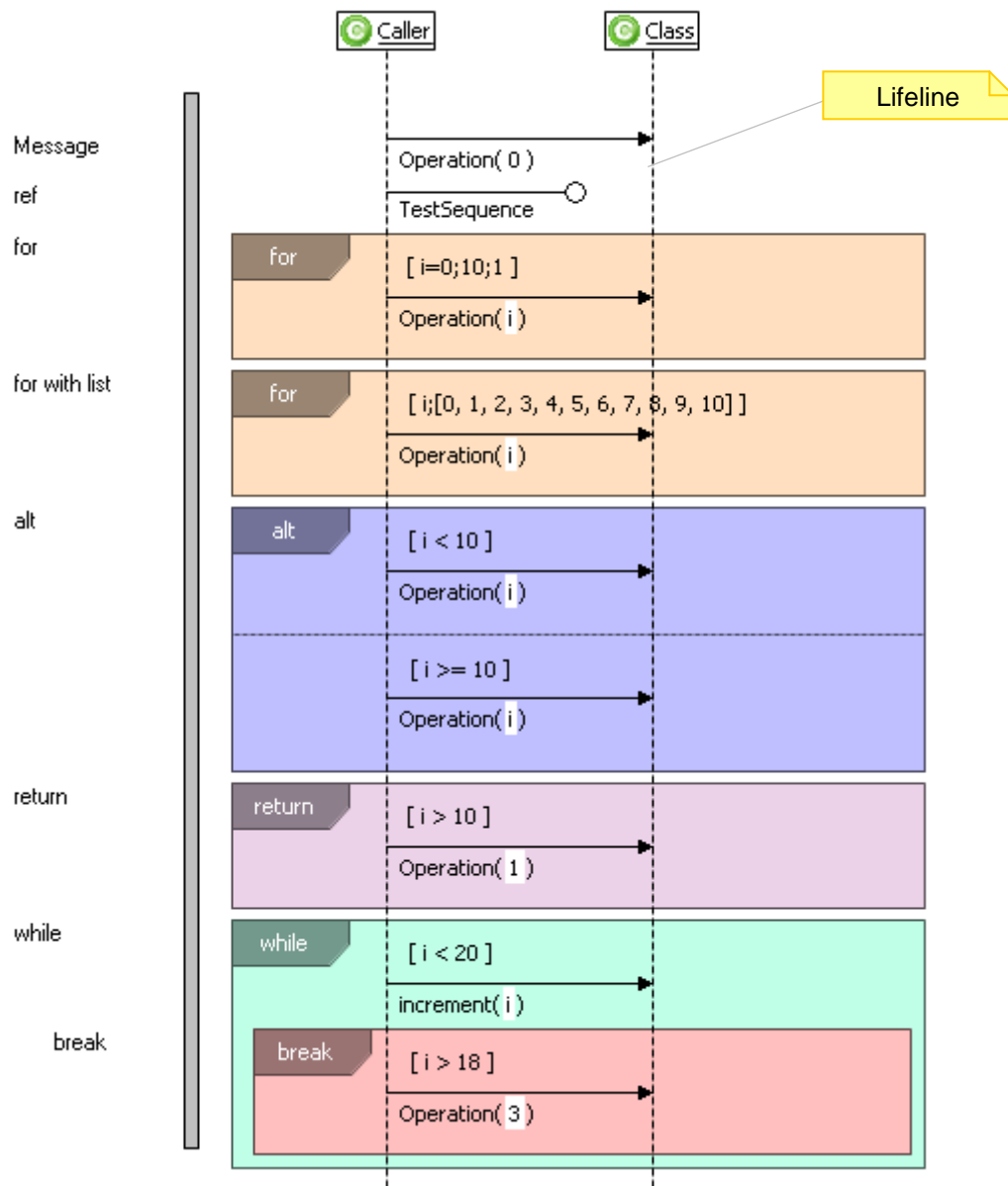
This section describes the control flow elements available in EXAM. The figure below shows an overview over those defined for TestActions and TestSteps.




Control flow elements in EXAM

6.2.1 Control flow elements in TestSequences

This section explains the control flow elements for TestSequences as introduced in 6.2.



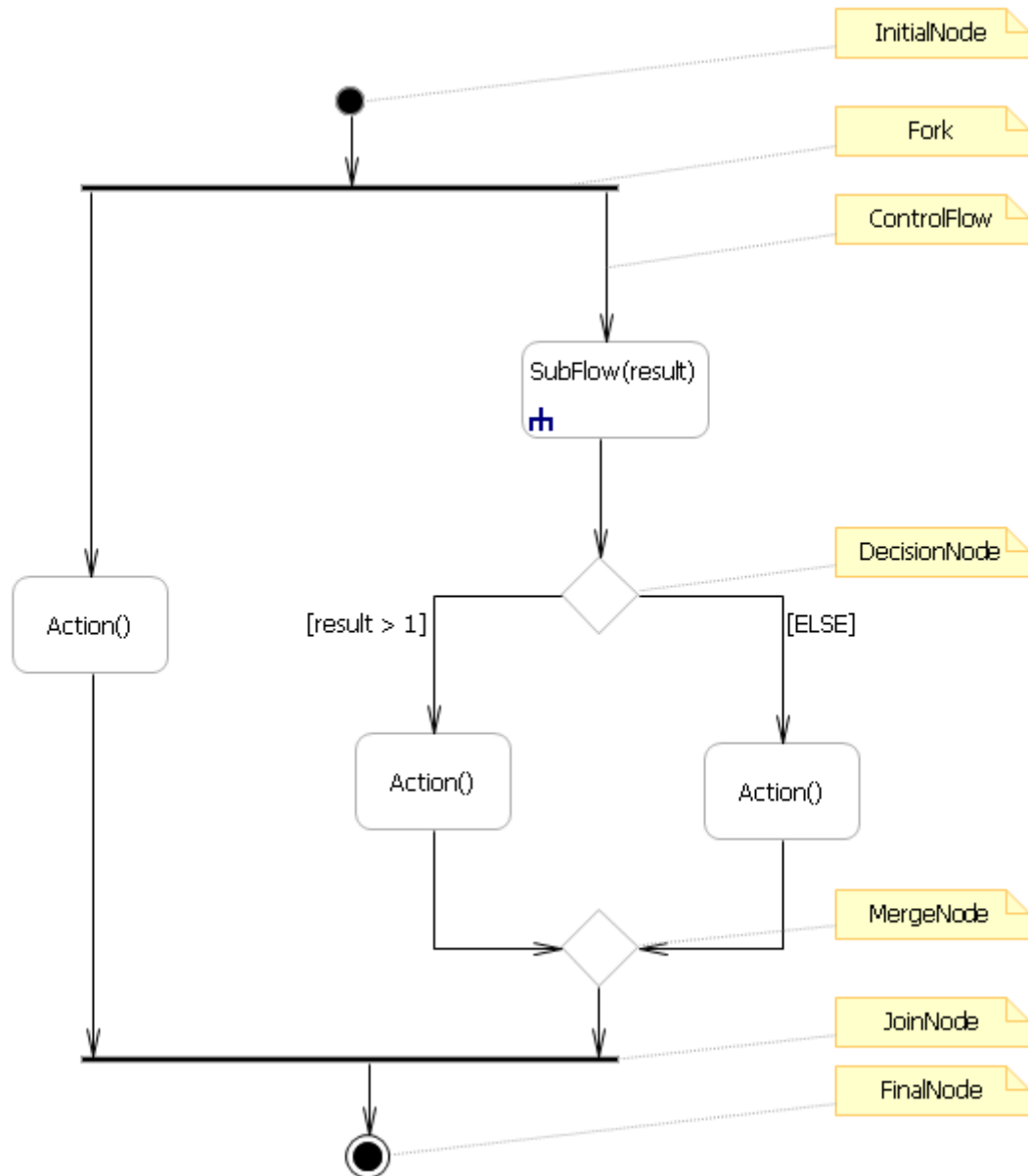
Control flow elements in TestSequences

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	73 of 104	
		Revision date:	7/29/2009	

Control flow element:	Description:
alt	Branch/alternative to implement IF-THEN-ELSE or SWITCH-CASE statements.
break	Expression to conditionally leave a loop block. It allows for defining a number of statements that should be executed before actually leaving the loop.
Lifeline	The life line represents the lifetime of an object in a sequence diagram.
loop - for	The for-loop can be used to execute a sequence of statements multiple times. The control the number of iterations, you can either specify an initial value, a condition and the step width or provide a list of values for the loop counter.
Loop - while	The while-loop can be used to execute a block of statements multiple times – as long as a given condition evaluates to true. This condition is checked before the block is executed.
Message	Calls an operation from an interface resp. an implementation class.
ref	Calls a TestFlow (TestSequence or TestActivity) from a sequence diagram (see also 6.11.1).
return	Conditional expression to leave a TestSequence. It allows for defining a number of statements that should be executed before actually leaving the TestSequence.

6.2.2 Control flow elements in TestActivities

This section explains the control flow elements for TestActivities as introduced in 6.2.



Control flow elements in TestActivities


Control flow element:	Description:
Action	Calls a TestAction from a TestActivity
ControlFlow	The ControlFlow is modeled with directed edges that define possible ways through the TestActivity.
DecisionNode	Node to conditionally branch (OR) within a TestActivity.
FinalNode	Node to leave a TestActivity
ForkNode	Node to parallelize (AND) the ControlFlow
InitialNode	Starting node/entry point of a TestActivity
InterruptibleAction	Calls an InterruptibleAction from a TestActivity
JoinNode	Node to synchronize the ControlFlow (after it has been parallelized).
MergeNode	Node to merge the ControlFlow (after it has been branched).
SubFlow	Call to a TestFlow in a TestActivity

6.3 Expressions in EXAM

This chapter describes expressions, which can be used in the test design. EXAM adapts to the nomenclature of the Python programming language [PYTHON]. The description only covers a small extent of what is possible in EXAM.

A. Expressions:

- Constants
e.g. 124; -24; 0; 1.23; 3.14e-10, ...
- Casts
e.g. `int(3.0)`; `str(1234)`; `float(2)`; ...
- Identifiers/Variables
e.g. A; a; _a; A1; ...
- Numeric operations
e.g. `-var`; `~var`; `'abc' + 'def'`; `1 - 6 * a` ...
- Relational operators
e.g. `a < 4`; `x == 's'`; `x != y`; ...
- Boolean operators
e.g. `var1 OR var2`; `var1 AND var2`
- dictionaries
e.g. `{'Max':0001, 'Paul':0002}`
- lists
e.g. `[1, 2, 3, 4]`
- tuples
e.g. `(5, 6, 7, 8)`

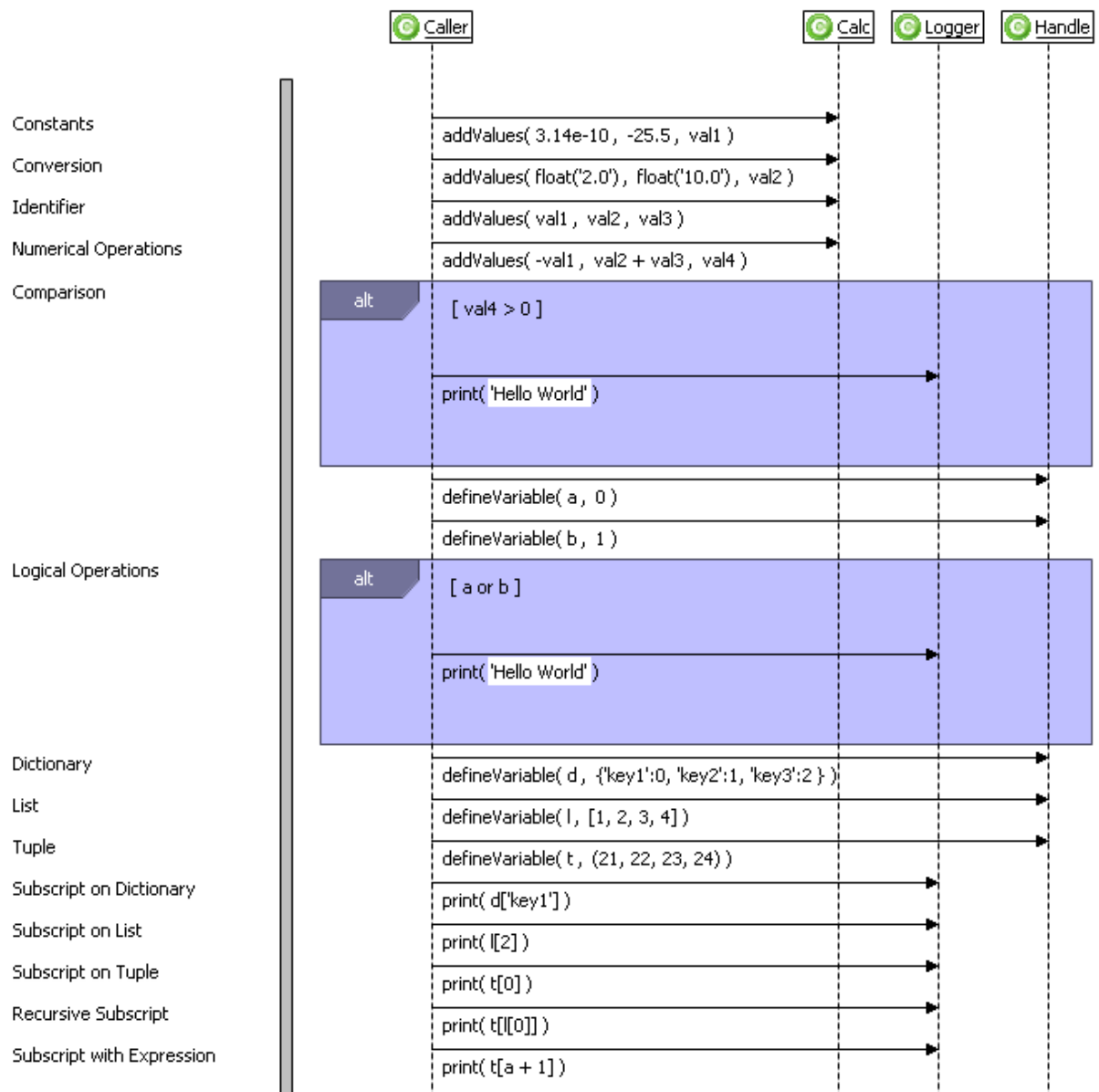
Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	76 of 104	
		Revision date:	7/29/2009	

- Subscripts (see B.)
- Operation calls
e.g. *A.getValue(ParClass.get())*
- Static attributes within the same class
- Enum attributes

Nesting expressions e.g. by parentheses is allowed.

B. Subscripts:


- Subscripts to access collection data types
e.g. *l[0], t[2], d['Max']*
- Nested subscripts
e.g. *a[b[4]]*
- Subscripts containing expressions
e.g. *a[1+i]*
- Slices are not supported
e.g. *[1:3], [1:10:2], [2:]*



Examples for expressions and subscripts in EXAM

Valid expressions/subscripts in ParameterSets:

- Constants
- Expressions that do not contain identifiers, except for casts
- Operation calls
- Enum attributes

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	78 of 104	
		Revision date:	7/29/2009	

Valid expressions/subscripts in operation parameters:

in:

- Expressions (see A.)
- Subscripts

out:

- Identifiers
- Subscripts (see B.)

inout

- Identifiers
- Subscripts (see B.)

Valid expressions/subscripts in GuardConditions:

alt:


- Identifiers
- Expressions of type int (top-level relational- or Boolean operators) (see A.)
- Operation calls
- Subscripts

loop – while:

- Identifiers
- Expressions of type int (see A)
- Operation calls
- Subscripts

loop – for:

- LoopVar
 - Identifiers
 - Subscripts (see B.)
- Initial value
 - Identifiers
 - Numeric expressions (see A.)
 - Operation calls
 - Subscripts
- Max value
 - Identifiers
 - Numeric expressions (see A.)
 - Operation calls
 - Subscripts
- Step
 - Identifiers
 - Numeric expressions (see A.)
 - Operation calls
 - Subscripts

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	79 of 104	
		Revision date:	7/29/2009	

loop – for with lists:

- LoopVar
 - Identifiers
 - Subscripts (see B.)
- ListExpression
 - Identifiers
 - Expressions of type list (see A.)
 - Operation calls
 - Subscripts

6.4 Implicit memory areas

The *centralTestHandle* is a special data type in EXAM. It can be used as a storage area with a defined functionality. The central test handle is divided into three dictionaries (see 6.1).

- *systemDictionary*
- *staticParameterDictionary*
- *dynamicParameterDictionary*

systemDictionary:


The *systemDictionary* is used to store system variables are defined by the test execution environment at the start or during the run time of the tests. The test designer only has read access to this information.

staticParameterDictionary:

The *staticParameterDictionary* is used to store global variables. The test designer can store variables and read them at any time during test execution.

dynamicParameterDictionary:

The *dynamicParameterDictionary* uses a stack mechanism. It is used to store parameter values and ParameterSet for TestFlows. The EXAM run time environment is responsible to keep the contents up-to-date with the test execution. When a new TestFlow is started, the associated ParameterSets are written to the dictionary. All values will be cleared when a TestFlow ends. The TestSequence accesses the variables using the get_ operations from the ParameterClasses while TestActivities use ParameterClassAttributes. All values are read-only.

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	80 of 104	
		Revision date:	7/29/2009	

6.5 (Multiple) Inheritance

EXAM allows using inheritance for most of its classes, ParameterSets and TestActivities. This section defines permissible and impermissible scenarios of multiple inheritance and inheritance in general.

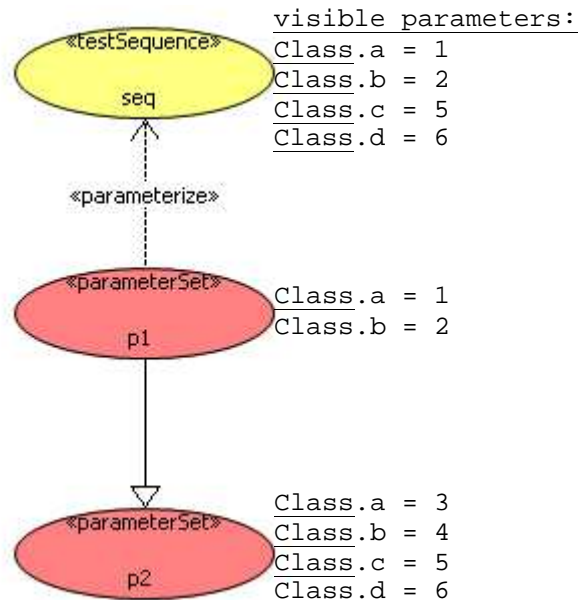
The table below provides an overview of whether single and/or multiple inheritance is allowed for particular EXAM objects.

EXAM object	Inheritance	Multiple inheritance
EventClass	YES	NO
EventMapping	NO	NO
FormalTestSpecificationClass	YES	YES
ImplementationClass	YES	YES
MappingClass	YES	YES
ParameterClass	YES	YES
ParameterSet	YES	YES
ShortNameClass	YES	YES
SystemConfiguration	YES	NO
TestActivity	YES	NO
TestSuite	YES	YES
VariableMapping	NO	NO

6.5.1 Inheritance between ParameterSets

This chapter explains in which cases inheritance between ParameterSet is permitted.

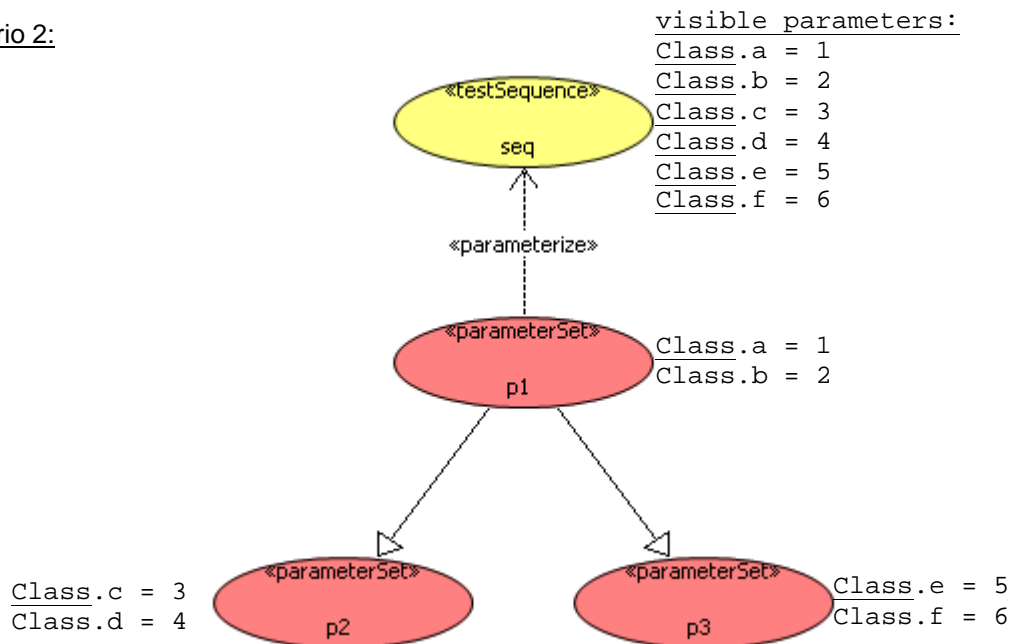
Scenario 1:



Inheritance between ParameterSets – scenario 1 (valid)

ParameterSet *p1* inherits parameters Class.a, Class.b, Class.c and Class.d from *p2*. Class.a and Class.b are overridden, because both *p1* and *p2* instantiate class *Class*, where *p1* is more specific from the TestSequence's point of view.

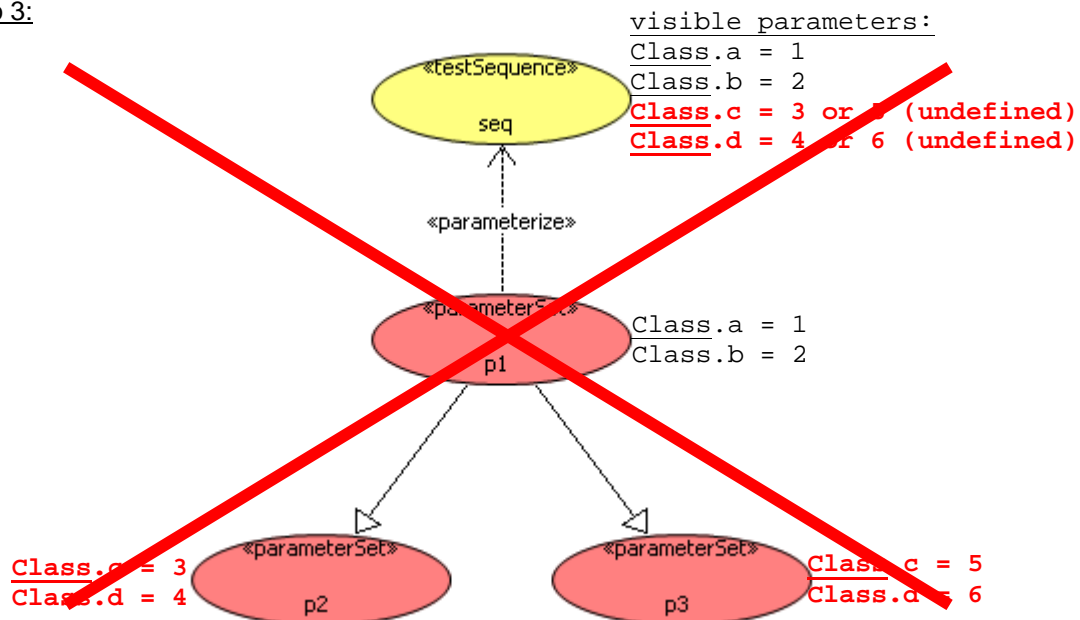
Scenario 2:



Inheritance between ParameterSets – scenario 2 (valid)

ParameterSet *p1* inherits parameters *Class.c* and *Class.d* from *p2*, and *Class.e* and *Class.f* from *p3*. *Class.a* and *Class.b* form *p1*'s own value pool.

Scenario 3:



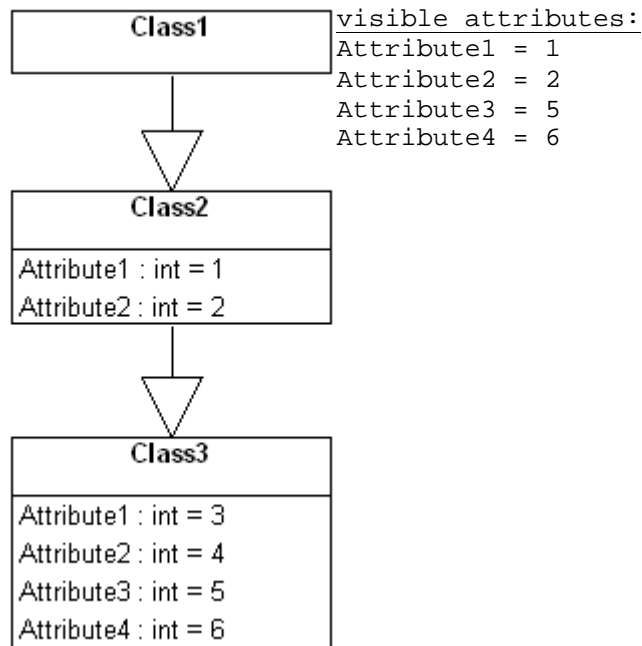
Inheritance between ParameterSets – scenario 3 (invalid)

Scenario 3 is invalid, because ParameterSet *p1* inherits parameters *Class.c* and *Class.d* from *p2* and *p3*. Thus, no unambiguous values can be calculated.

6.5.2 Inheritance of attributes

This chapter explains in which cases inheritance of class attributes is permitted.

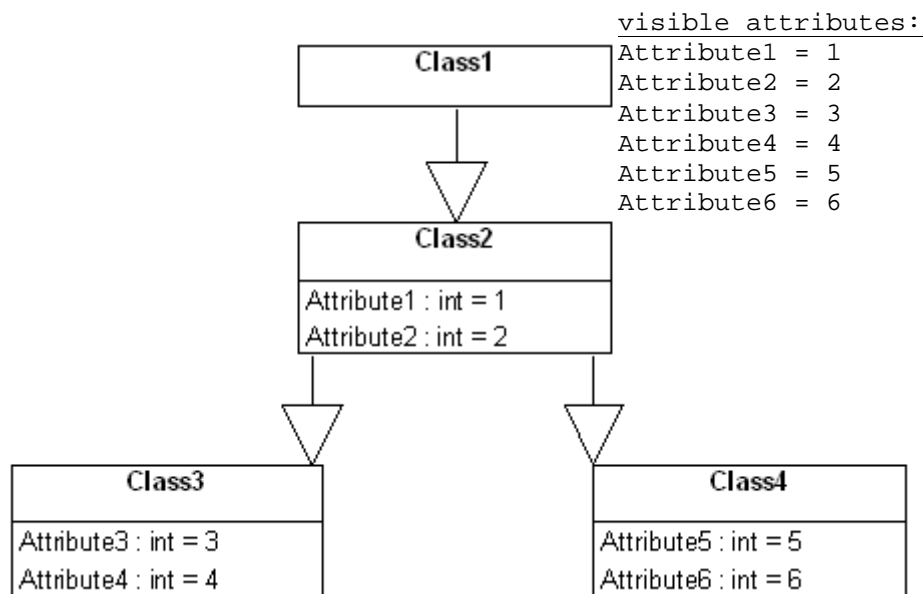
Scenario 1:



Inheritance of attributes – scenario 1 (valid)

Class2 inherits attributes *Attribute1-4* from *Class3*. *Attribute1* and *Attribute2* are overridden by *Class2*, because they are more specific from *Class1*'s point of view.

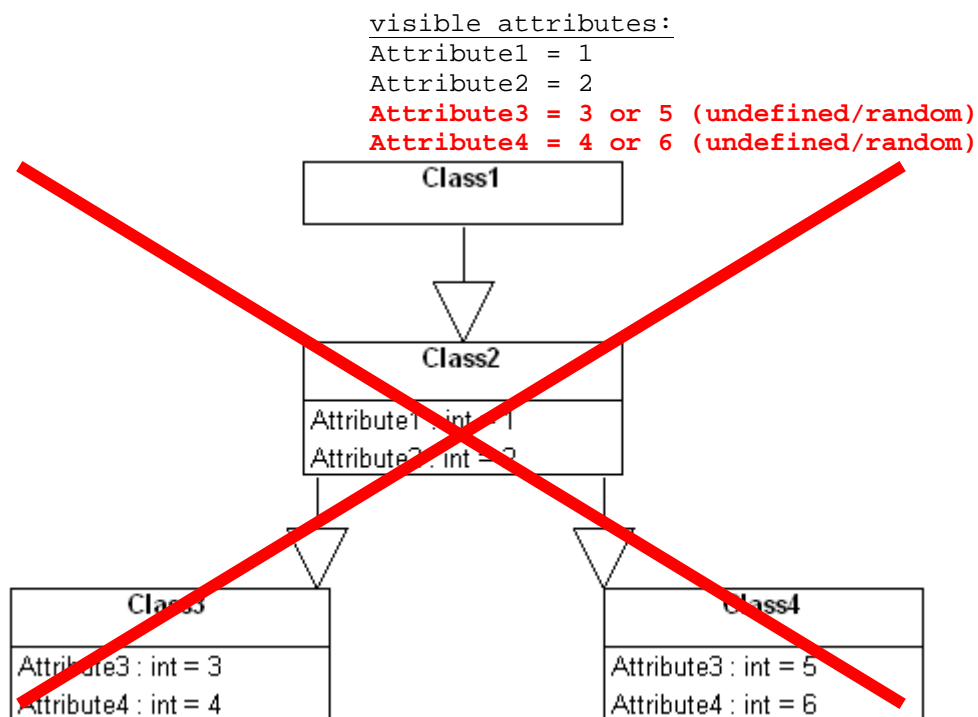
Scenario 2:



Inheritance of attributes – scenario 2 (valid)

Class2 inherits *Attribute3* and *Attribute4* from *Class3* and *Attribute5* and *Attribute6* from *Class4*. *Attribute1* and *Attribute2* form *Class2*'s own value pool.

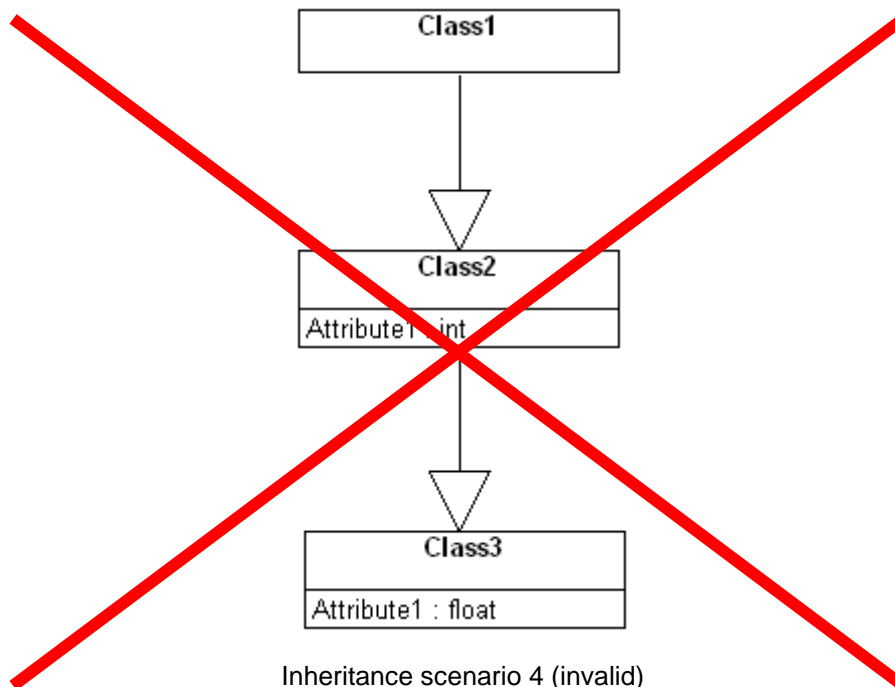
Scenario 3:



Inheritance of attributes – scenario 3 (invalid)

Scenario 3 is invalid, because Class *Class1* inherits attributes *Attribute3* and *Attribute4* from *Class3* and *Class4*. Thus, no unambiguous values can be calculated.

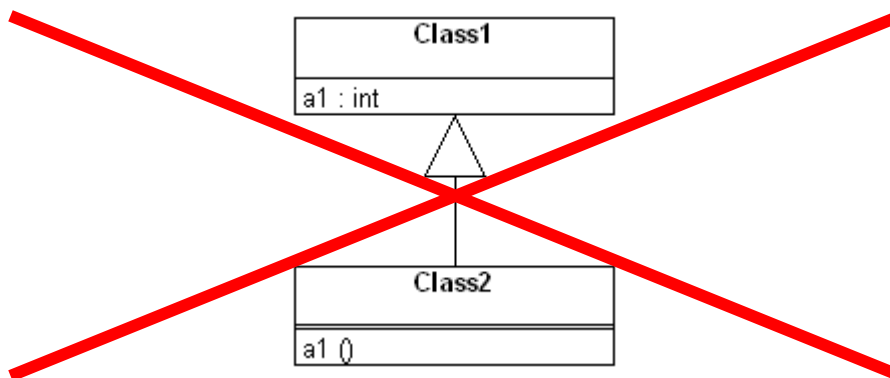
Scenario 4:



Inheritance scenario 4 (invalid)

Scenario 4 is invalid, because attributes with the same name but different data types must not exist within an inheritance hierarchy.

Scenario 5:



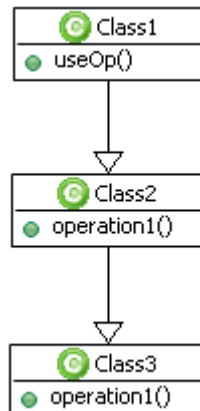
Inheritance scenario 4 (invalid)

Scenario 5 is invalid, because an attribute and an operation with the same name must not exist within an inheritance hierarchy.

6.5.3 Inheritance of operations

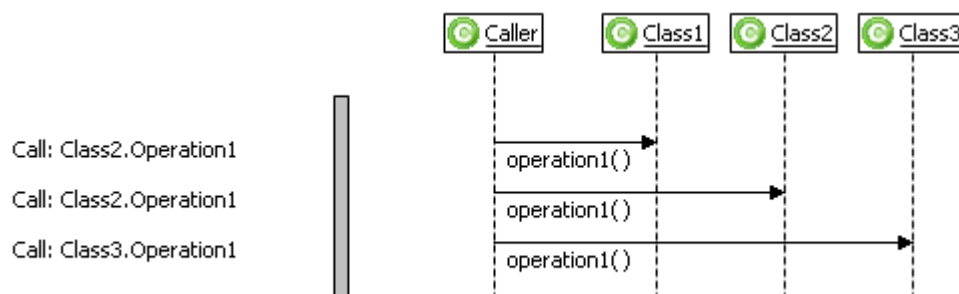
This chapter explains in which cases inheritance of class operations is permitted.

Scenario 1:



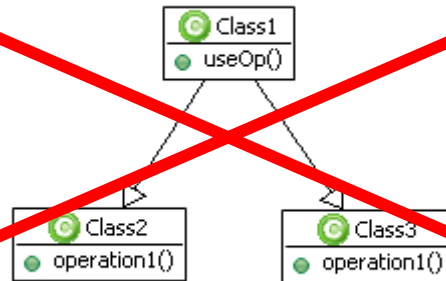
Inheritance of operations – Scenario 1 (valid)

Class1 inherits operation *operation1* from *Class2*, which overrides *operation1* from *Class3*. The operation can be called on Classes 1, 2 and three, where the implementation from *Class2* is used for the first two cases while the implementation from *Class3* is used for the latter case.



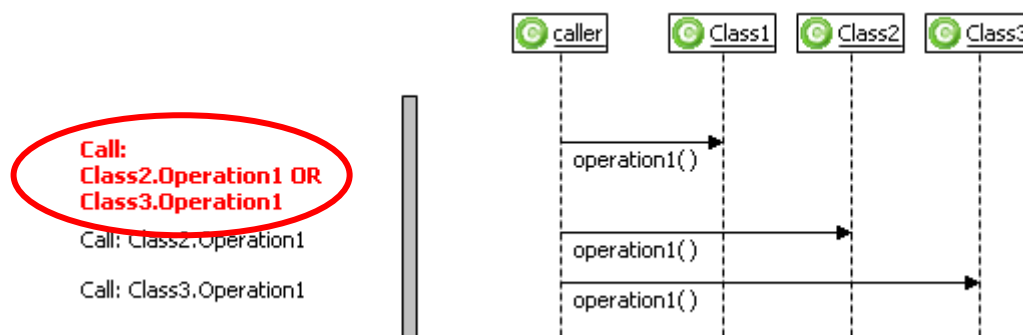
Calling operations – Scenario 1

Scenario 2:



Inheritance of operations – Scenario 2 (invalid)

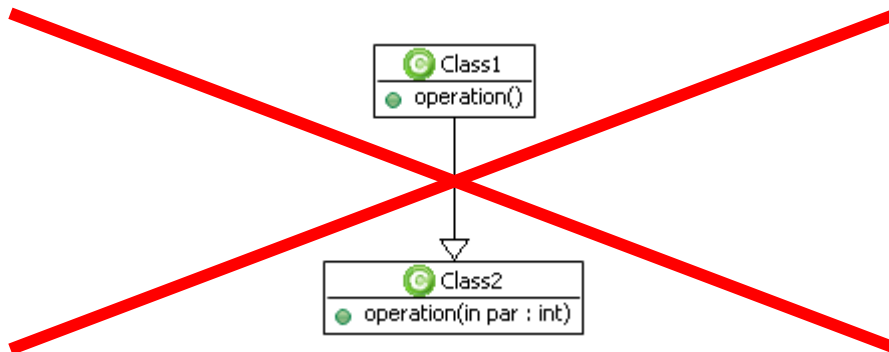
Scenario 2 is invalid, because Class *Class1* inherits operation *operation1* from *Class2* **and** *Class3*. Thus, it cannot be determined which operation to call.



Calling operations – Scenario 2

The first operation call is not allowed, because it cannot be determined which implementation of *operation1* should be used. Thus, inheriting the same operation from multiple sources is not allowed.

Scenario 3:



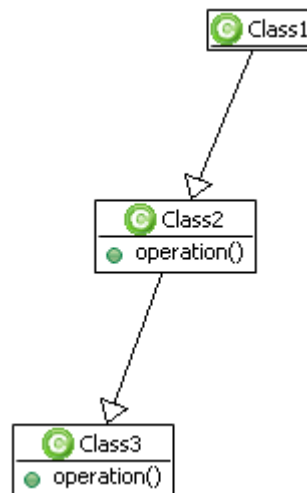
Inheritance of operations – Scenario 3 (invalid)

Scenario 3 is invalid, because operations with the same name but different signature must not exist within an inheritance hierarchy.

6.6 Operation overloading

This chapter explains in which cases overloading of operations is permitted.

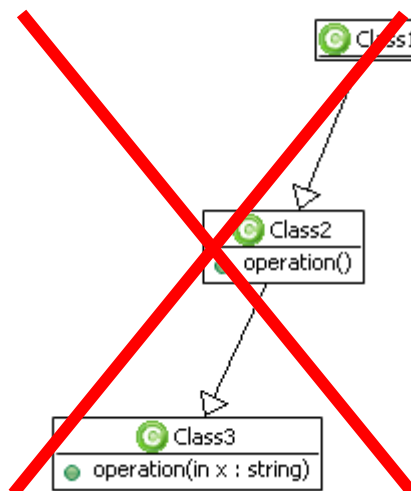
Scenario 1:



Operation overloading – Scenario 1 (valid)

Overloading operations with the same signature within an inheritance hierarchy is permitted in EXAM.

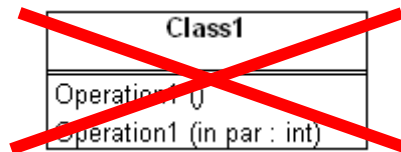
Scenario 2:



Operation overloading – Scenario 2 (invalid)

Overloading operations with different signatures within an inheritance hierarchy is not permitted in EXAM.

Scenario 3:



Operation overloading – Scenario 3 (invalid)

Overloading operations with different signatures within a single class is not permitted in EXAM.

6.7 Comments

Similar to writing source code it is possible to comment out steps in sequence diagrams. This is done by assigning the «comment» stereotype to the particular step.

6.8 Signature

Operations in EXAM have the following notation:

OPERATION_NAME(<PARAMETER_LIST>)

EXAM does not use return parameters. Results are returned by reference using out resp. inout parameters. The signature of a parameter is defined as follows:

<TYPE> PARAMETER_NAME[<MECHANISM>] = [<VALUE>]

Possible mechanisms are:

- **in:** The value is handed over from the caller to the callee.
- **out:** The value is handed over from the callee to the caller.
- **inout:** The value is handed over from the caller to the callee and, possibly with changed values, back again at the end of the operation.

Valid data types for the **<TYPE>** field have already been explained in chapter 6.1. For parameters with the in mechanism, it is possible to define default values (**<VALUE>**). This allows you, to call the operation without specifying a concrete value for the particular parameter. In that case, its default value is used. Parameters with default value have to be defined at the end of the **<PARAMETER_LIST>**. Additionally, if you choose not to provide concrete values for one or more parameters in an operation call, these have to be **at the very end** of the parameter list with no gaps between them. Otherwise it would not be possible to match given parameter values to the actual parameters. However, out-parameters may be defined after parameters with default values.

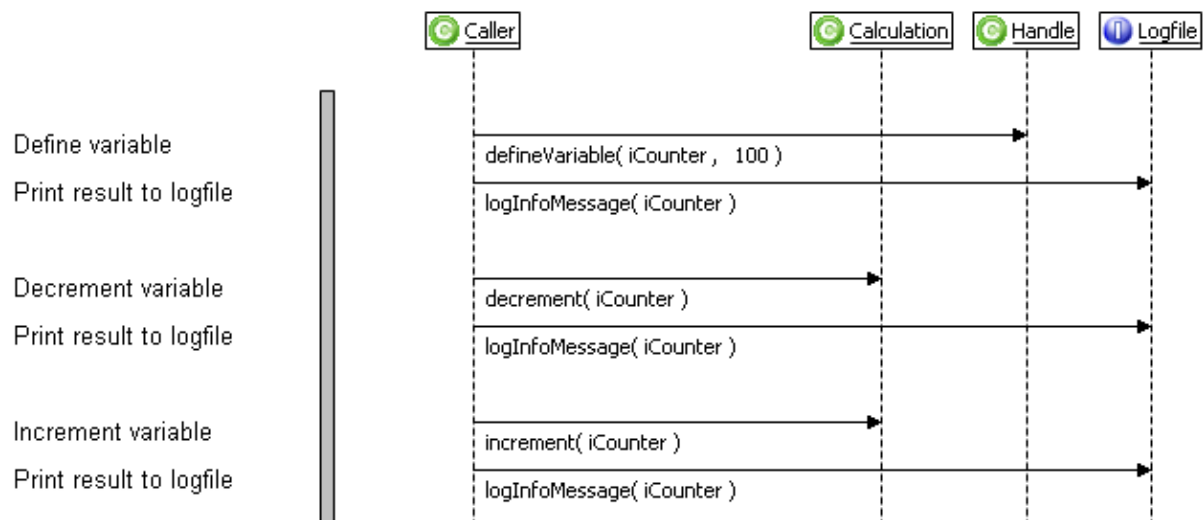
6.9 Naming conventions

The naming conventions listed below apply in EXAM.


- Identifier must only be composed from a..z, A..Z, 0..9 and _ (underscore) characters. They must not start with a digit. Other characters, in particular white spaces, umlauts, "ß" or special characters are not allowed.
- The names of EXAM objects must be unique within their particular structural unit, i.e. there must not be multiple objects with the same name and the same parent object. For example attributes and operations within one single class or objects in a particular package need to have different names.

6.10 Caller class

Messages to class instances on sequence diagrams have to origin from the *Caller* class. This provides a uniform appearance and good readability of the diagrams.



Example for using the *Caller* class

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	92 of 104	
		Revision date:	7/29/2009	

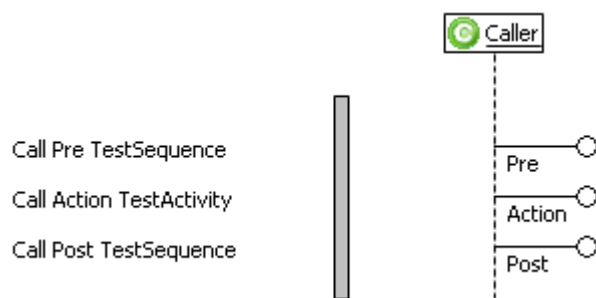
6.11 Deviations from the UML standard

The implementation of EXAM in a software tool requires some deviations from or extensions to the UML. This can be to add non-existing description elements or to improve usability of the tool.

6.11.1 Including TestFlows in sequence diagrams

The call of test flows (TestSequences and TestActivities) within sequence diagrams is done using interaction frames as described in section 5.1.1.3.3. For legacy reasons, in EXAM there is an alternative presentation form.

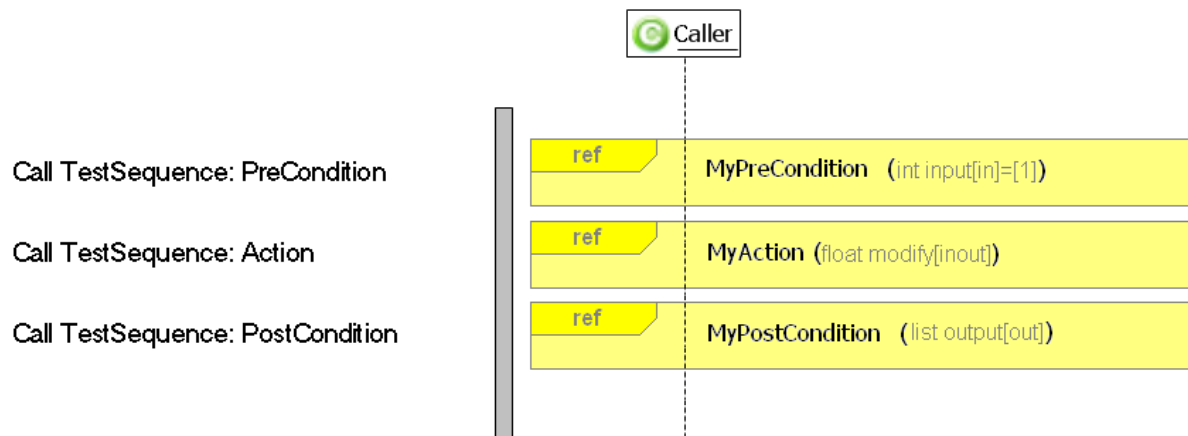
The TestFlow calls are modeled as a special message from the *Caller* class (see 6.10) to their particular use case. Since use cases cannot be instantiated on a sequence diagram, the message ends in a circle. It is said that the sequence diagram “includes” the use case. From EXAM version 3.x, this notation is replaced by the standard one using interaction frames (see 5.1.1.3.3.1).



Sequence diagram that includes three TestFlows

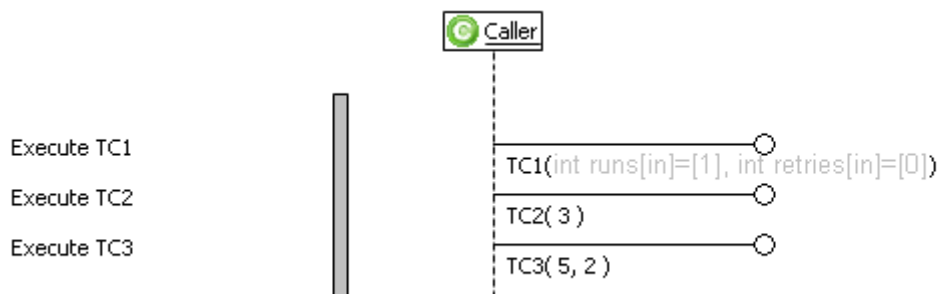
6.11.2 Signatures of TestFlow includes

EXAM currently offers two ways to model TestFlow includes in sequence diagrams; using interaction frames (see 5.1.1.3.3) and the alternative solution described in section 6.11.1. In both versions, parameters can be passed to the included TestFlows. Their signatures are equal to those for operations as introduced in section 6.8.



Signatures of TestFlow includes

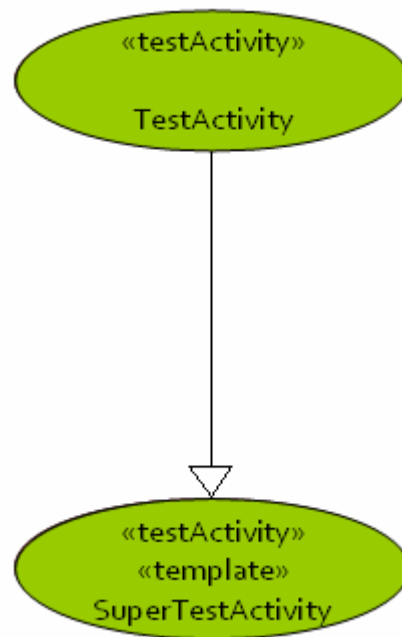
Passing parameters to TestFlows is used e.g. when invoking TestCases from a TestGroup. This allows for passing the *runs* (number of runs) and *retries* (number of retries in case of errors) parameters. The figure below shows signatures using the alternative notation.




Invocation of TestCases from a TestGroup

6.11.3 Template concept for TestActivities

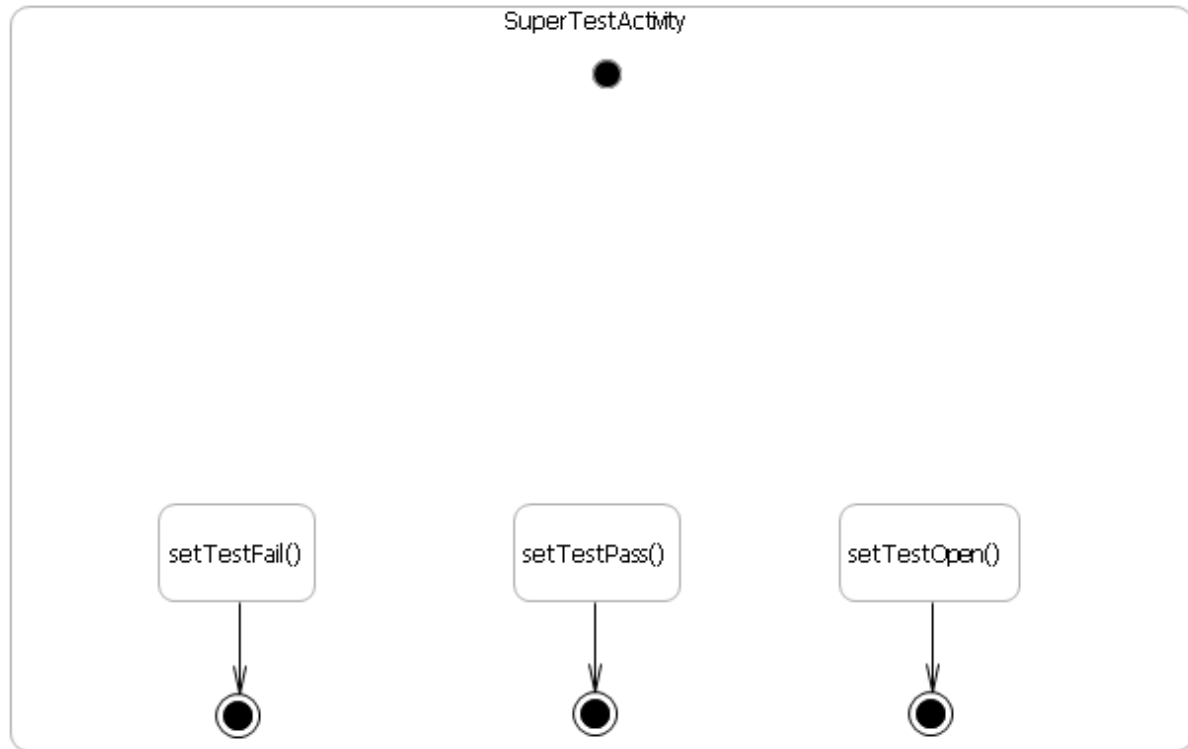
EXAM allows for defining inheritance relations between TestActivities. This does not involve inheritance in common sense but rather some sort of “inheritance of model elements”. The super TestActivity is a template that provides model elements to derived TestActivities.



Inheritance between TestActivities

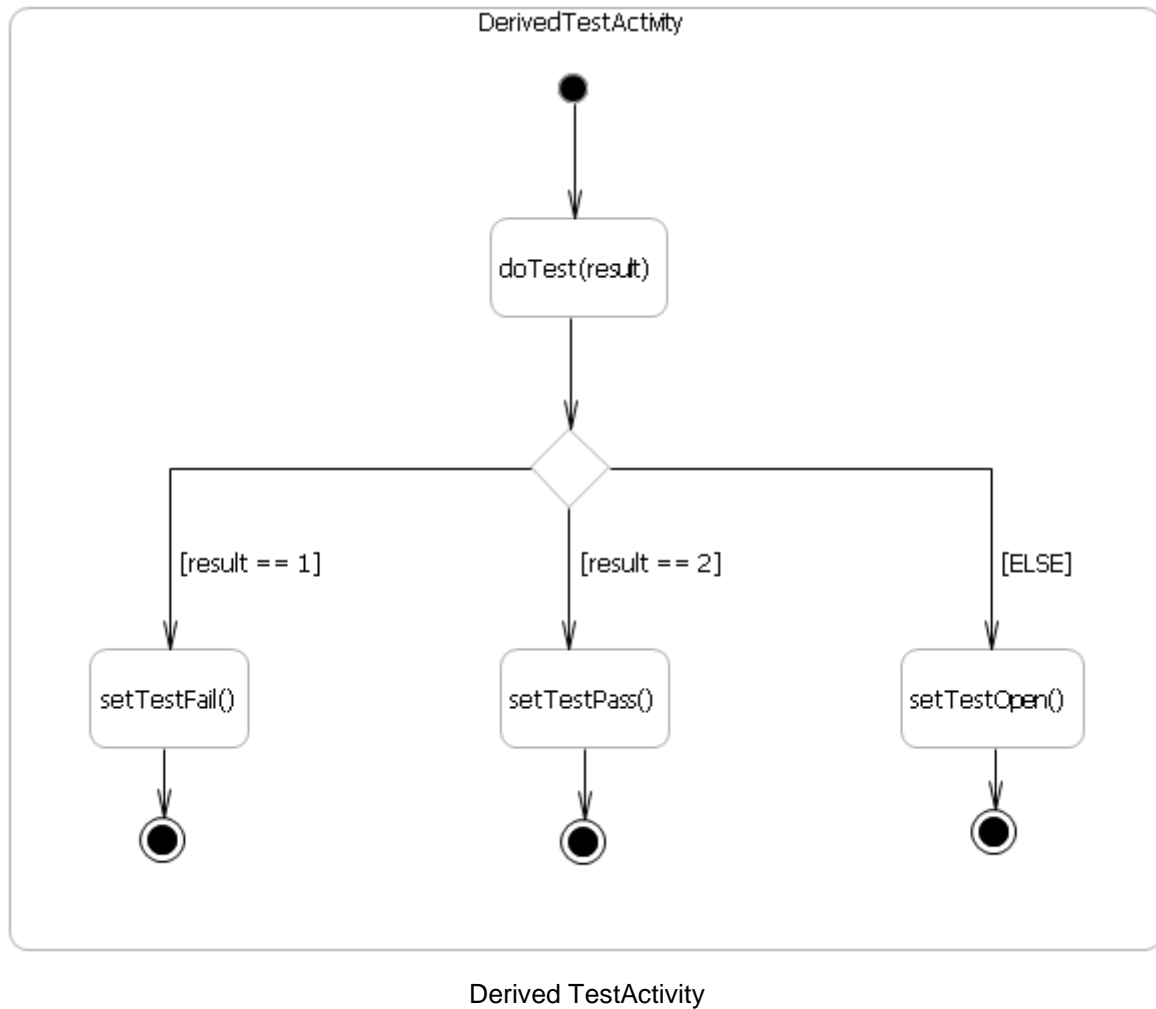
Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	95 of 104	
		Revision date:	7/29/2009	

Super TestActivities are labeled with the «*template*» stereotype. They are not necessary complete. The figure below shows an example for an incomplete TestActivity.



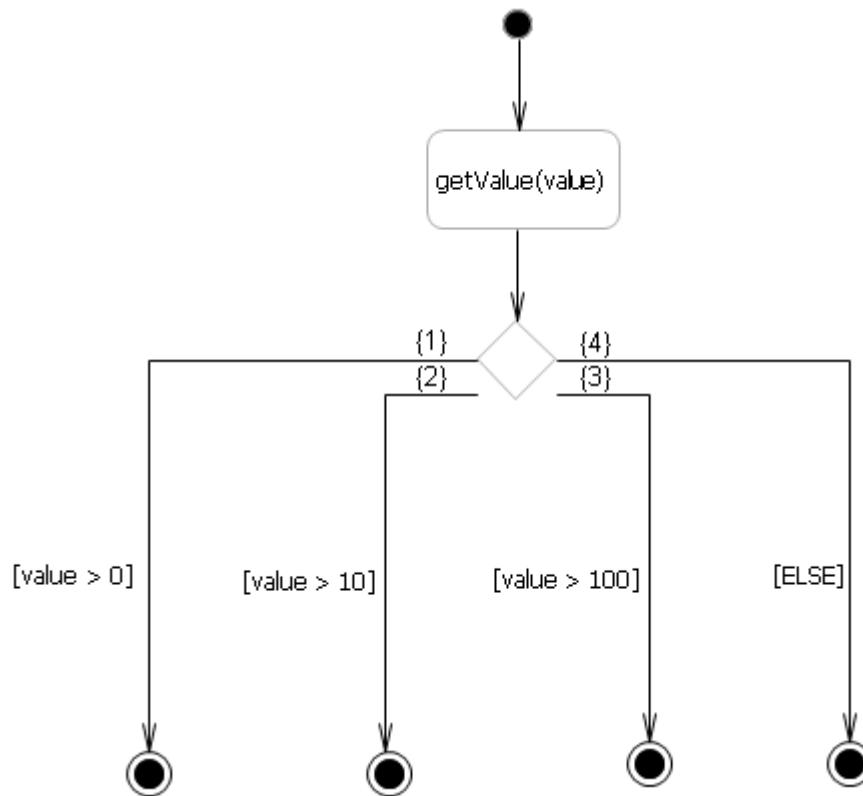
Incomplete super TestActivity

Derived TestActivities contain all elements from the template. They can be visually changed or repositioned but cannot be deleted.



6.11.4 Indices in TestActivities

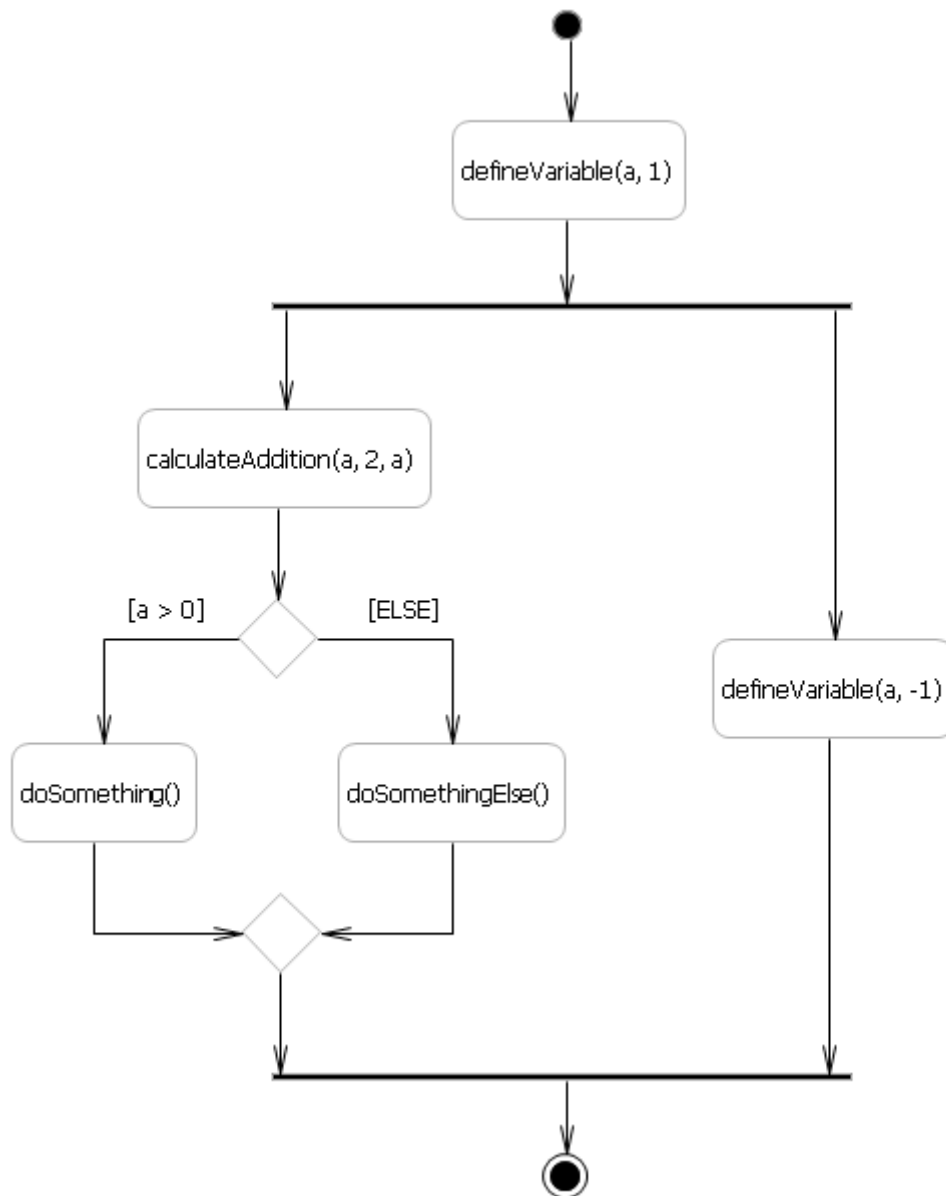
When using decision nodes in TestActivities, EXAM offers the possibility to explicitly define the order in which GuardConditions are evaluated. This is done by specifying indices for the outgoing edges of such a node. These indices can be arbitrary integers. Evaluation is done in ascending order. In the figure below, the GuardConditions are evaluated from the left to the right.



Using indices in TestActivities

6.11.5 Synchronization in TestActivities

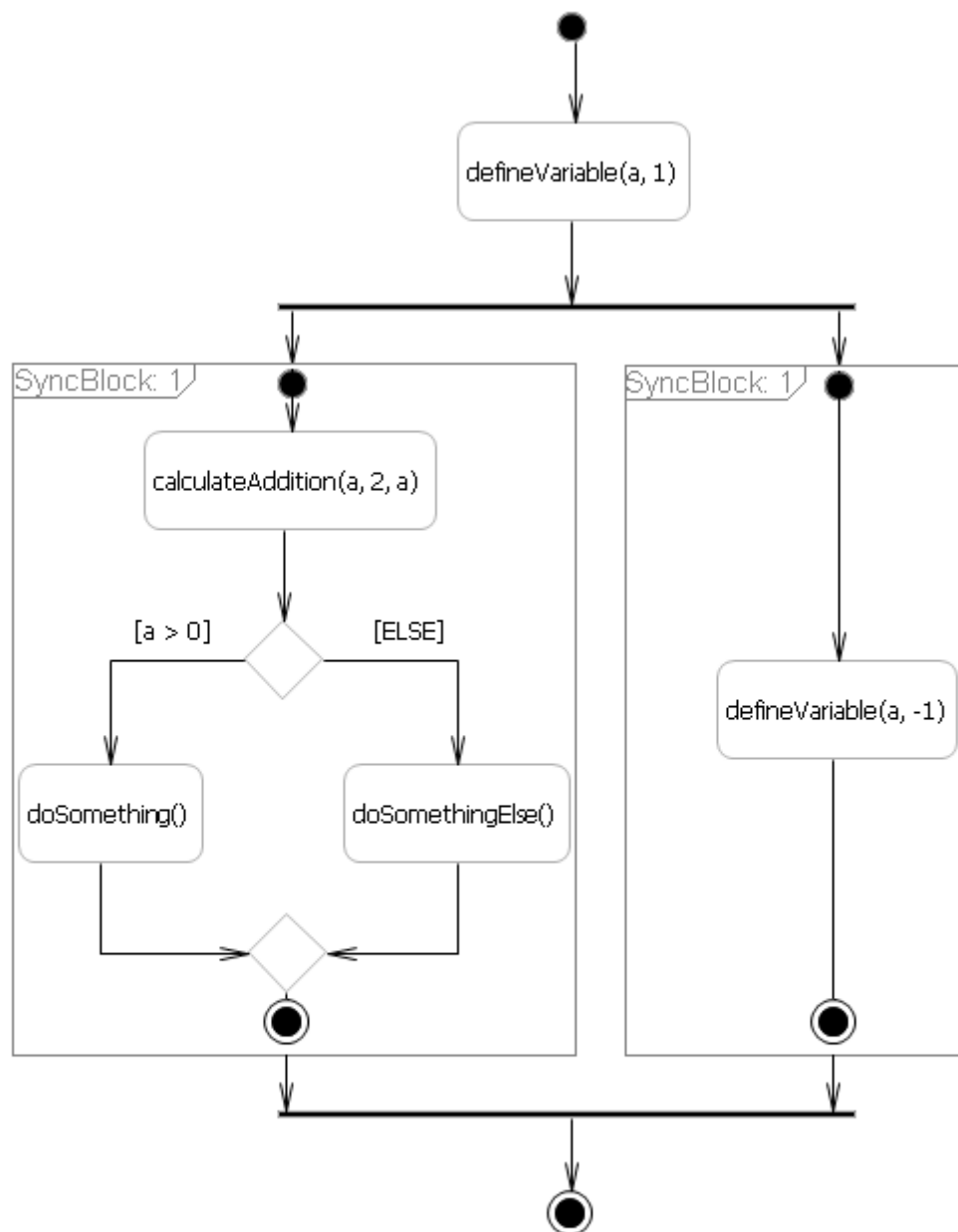
TestActivities allow parallelizing the ControlFlow. From a technical point of view this means that multiple threads are used which leads to a semi-parallel execution (because the hardware might not be able to execute all of them in parallel). This can lead to unwanted side effects.



Example for multithreading effects

In this example, both paths write to variable *a* and thus change its value. Depending on the actual execution order and the position where a possible context switch between those two threads may occur, variable *a* may have a different value which then leads to different execution paths for the decision node.

This can be prevented by using *SyncBlocks*.

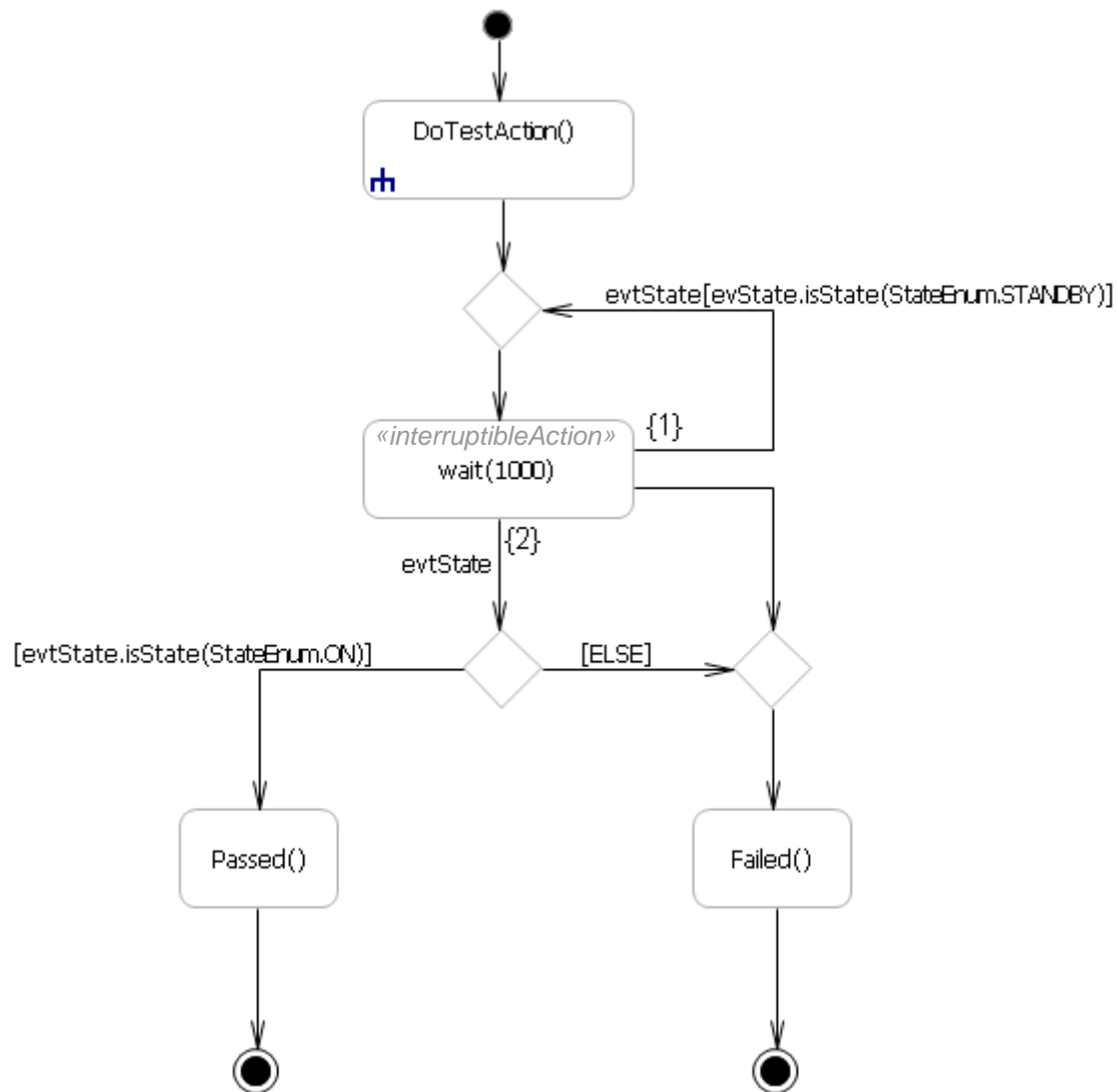


Using *SyncBlocks* in TestActivities

SyncBlocks are defined at least in pairs and have an integer ID that identifies connected blocks. While executing one *SyncBlock*, it is guaranteed that no other thread will enter a *SyncBlock* with the same ID. It is not determined which of the connected *SyncBlocks* will be executed first.

6.11.6 InterruptibleActions


Besides the *waitForEvent* operation, EXAM provides another concept for processing events within a TestActivity: InterruptibleActions. InterruptibleActions are a special form of a TestAction and are labeled with the «*interruptibleAction*» stereotype. The following figure shows an example for processing events using InterruptibleActions.



Using InterruptibleActions in TestActivities

InterruptibleActions allow for creating a new token in a TestActivity on the first occurrence of one of the events defined at the outgoing edges. The GuardCondition of that edge (denoted in square brackets) must be fulfilled; evaluation of the GuardConditions is done in the order specified by the indices provided (denoted in curly braces, see 6.11.4). The original Token is deleted immediately and only one new token will be created. InterruptibleActions must only have parameters with mechanism in, not out or inout (see 6.8). The syntax for outgoing edges of an Interruptible action is:

< EVENTCLASS_NAME>[<GUARD_CONDITION>]

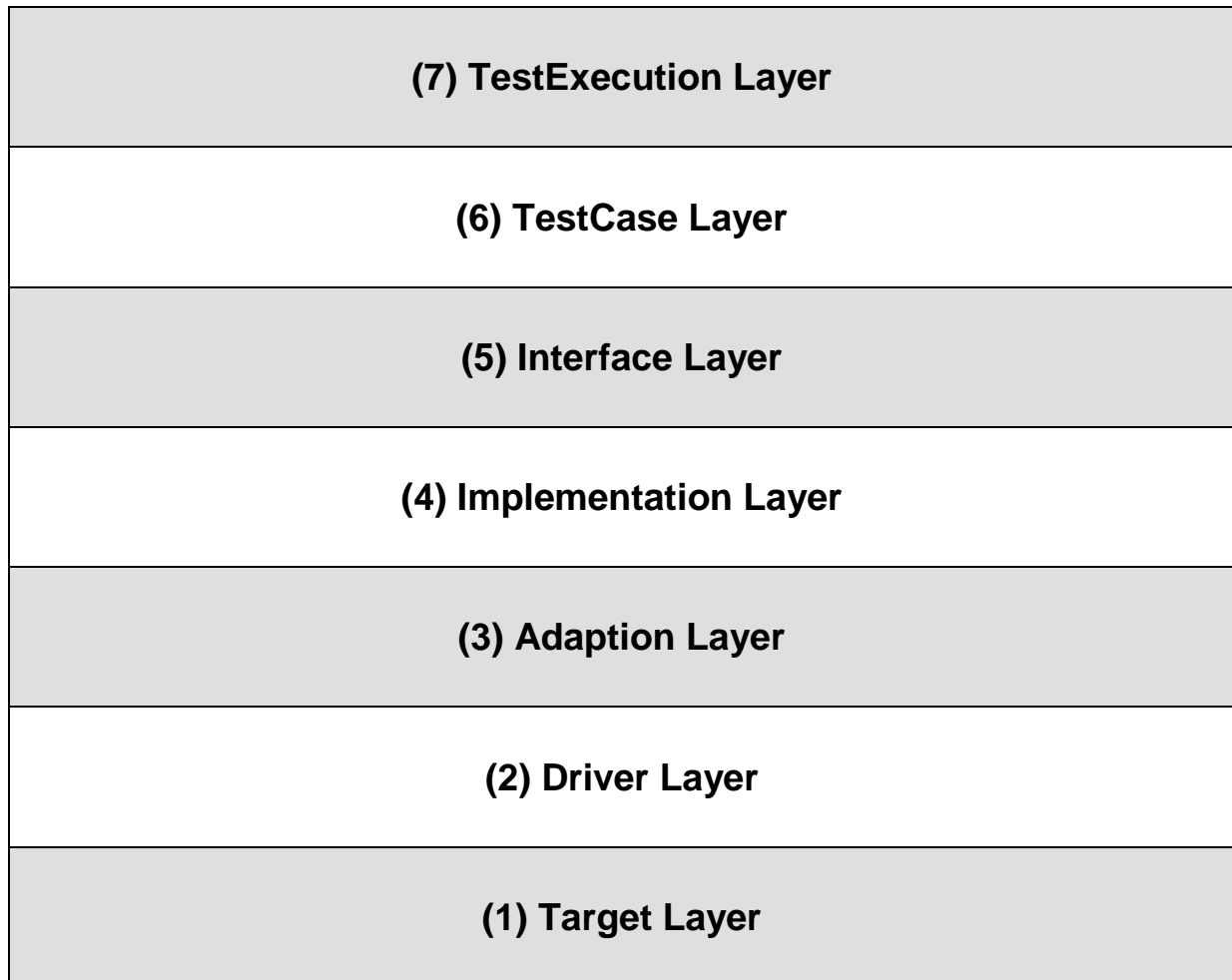
Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	101 of 104	
		Revision date:	7/29/2009	


For the diagram above, the following semantics apply:

The ControlFlow enters the InterruptibleAction *wait*. A waiting function is carried out. If the *wait* operation terminates without an event of type *evtState* occurring, the token leaves the TestAction on the lower right edge leading to *Failed()*. However, if the *evtState* event occurs, the GuardConditions on the outgoing edges are evaluated in the order defined by their particular index. First, the edge with index {1} is checked. If the value of the event is STANDBY (for enumerations see 6.1), the TestAction is left on edge {1} and the *wait* operation is repeated. Otherwise, the GuardCondition at edge {2} is evaluated. If it is true, the new token is created at that edge and the event is evaluated at the DecisionNode (see 6.2.2). As soon as the InterruptibleAction is left through output {1} or {2}, the *wait* operation is terminated.

7 EXAM Layer Model

In this section, the architecture of EXAM is explained using a layer model. EXAM is hierarchical, i.e. the layers build up on each other



Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	103 of 104	
		Revision date:	7/29/2009	

(1) Target Layer

The Target Layer is any kind of test system that should be automated with EXAM. Examples include HiL test benches of various manufacturers, diagnostic systems and power supplies as well as software systems such as CANoe, CANape, INCA, etc.

(2) Driver Layer

The Driver Layer is comprised of software interfaces that are required to control the components from the Target Layer. Usually, these drivers are provided by the manufacturers of the particular hardware system.

(3) Adaption Layer

The Adaption Layer integrates the Driver Layer into EXAM's software environment. This is done by e.g. creating wrappers for the drivers in the target programming language.

(4) Implementation Layer

The Implementation Layer consists of the ImplementationClasses in EXAM. The operations can be defined using sequence or activity diagrams or program code. Layers (4) and (5) actually are fairly complex, but from the conceptual point of view only the two of them exist.

(5) Interface Layer


The Interface Layer decouples the FormalTestSpecification from the TestCaseImplementation. It consists of FormalTestSpecificationClasses that are implemented by classes from the Implementation Layer. This layer allows for portability from one test system to another.

(6) TestCase Layer

The TestCase Layer is used to compose TestCases, ideally only using classes from the Interface Layer.

(7) TestExecution Layer

The TestExecutionLayer groups TestCases to ensure an effective and safe test execution on the test bench and to separate competences within an EXAM project.

Concept paper	EXAM EXtended Automation Method	Department:	I/EE-65	
		Page:	104 of 104	
		Revision date:	7/29/2009	

8 Bibliography

[EXAM DOC]	Documentation available from the EXAM core model
[EXAM MM]	Dr. Frank Derichsweiler (Audi I/EE-33), Gerhard Kiffe (Audi I/EE-33): „EXAM-Metamodell“ (“EXAM meta model”); UML model; Ingolstadt; 2005
[ITS]	Michael Schläfer (MicroNova AG): „ITS-Konzeptpapier“ (“ITS concept paper”); Vierkirchen; 2004
[KONV]	Audi I/FP-1: “Konventionenhandbuch - V2.3” (“Conventions guide – V2.3”); Ingolstadt; 2005
[PM]	Gerhard Kiffe (Audi I/EE-33): „EXAM-Prozessmodell“ (“EXAM process model”); ARIS model; Ingolstadt; 2006
[PYTHON]	http://www.python.org – Official homepage of the Python programming language
[UML2.0]	Bernd Oestereich: „Die UML 2.0 Kurzreferenz für die Praxis“ (“The UML 2.0 Quick Reference Guide”); Oldenburg; München, Wien; 2004