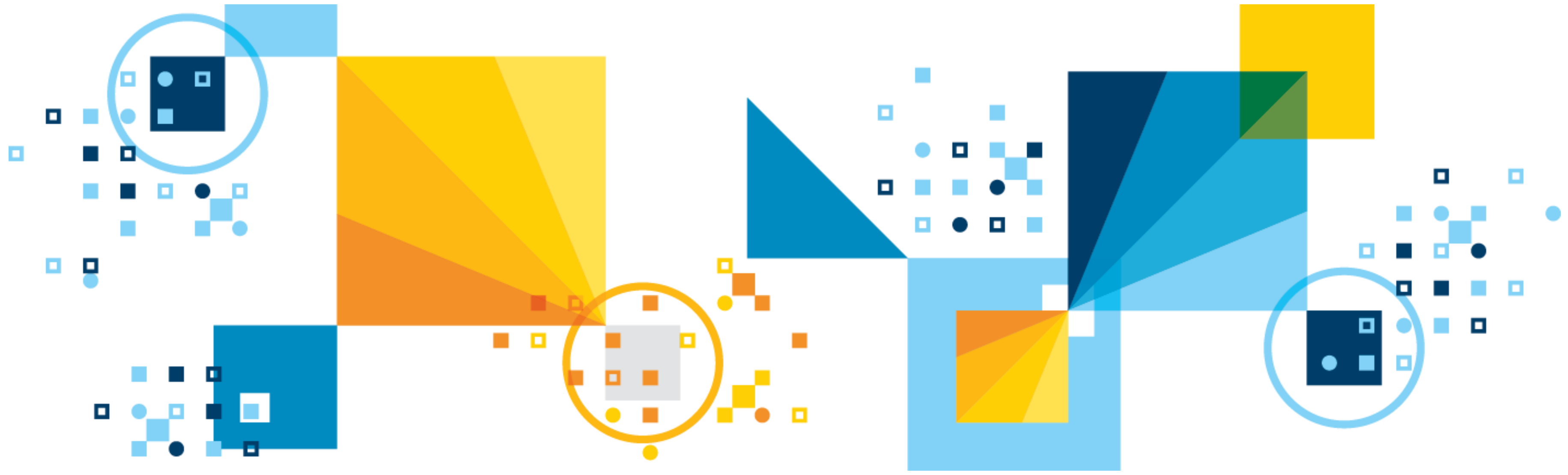
What is Model-Based Testing … and how do I get started?

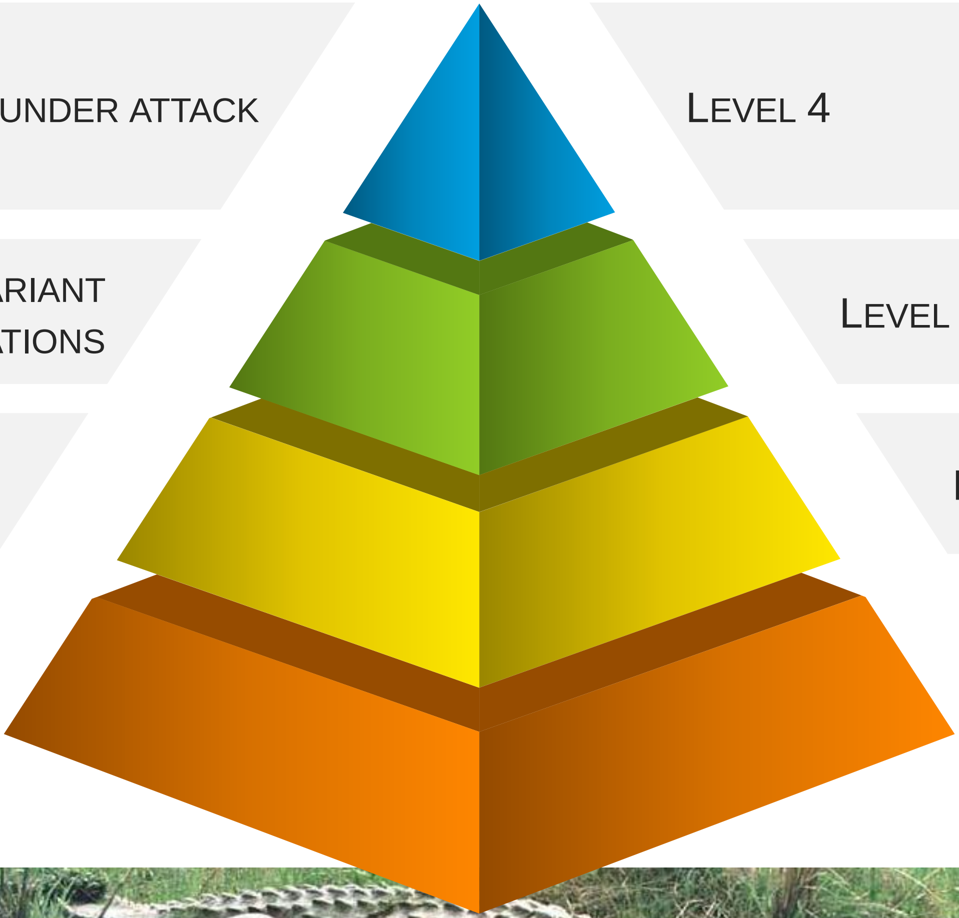
Bruce Powel Douglass, Ph.D.
Chief Evangelist, Global Technology Ambassador
IBM Internet of Things (IoT)
bruce.douglass@us.ibm.com
Twitter: @IronmanBruce
Website: : www-01.ibm.com/software/rational/leadership/thought/brucedouglass.html



Thanks to Udo Brockmeyer of BTC Embedded Systems AG for permission to use some of his material on Test Conductor

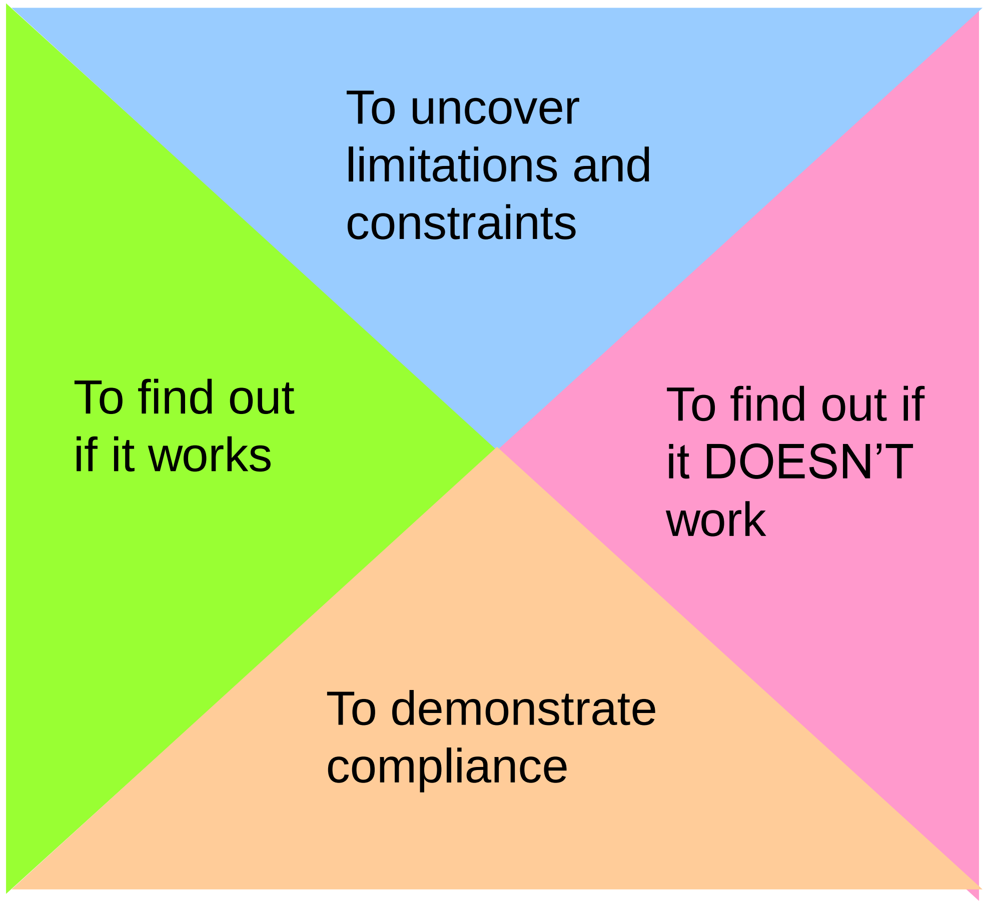
All code is guilty, until proven innocent.

Levels of Correctness

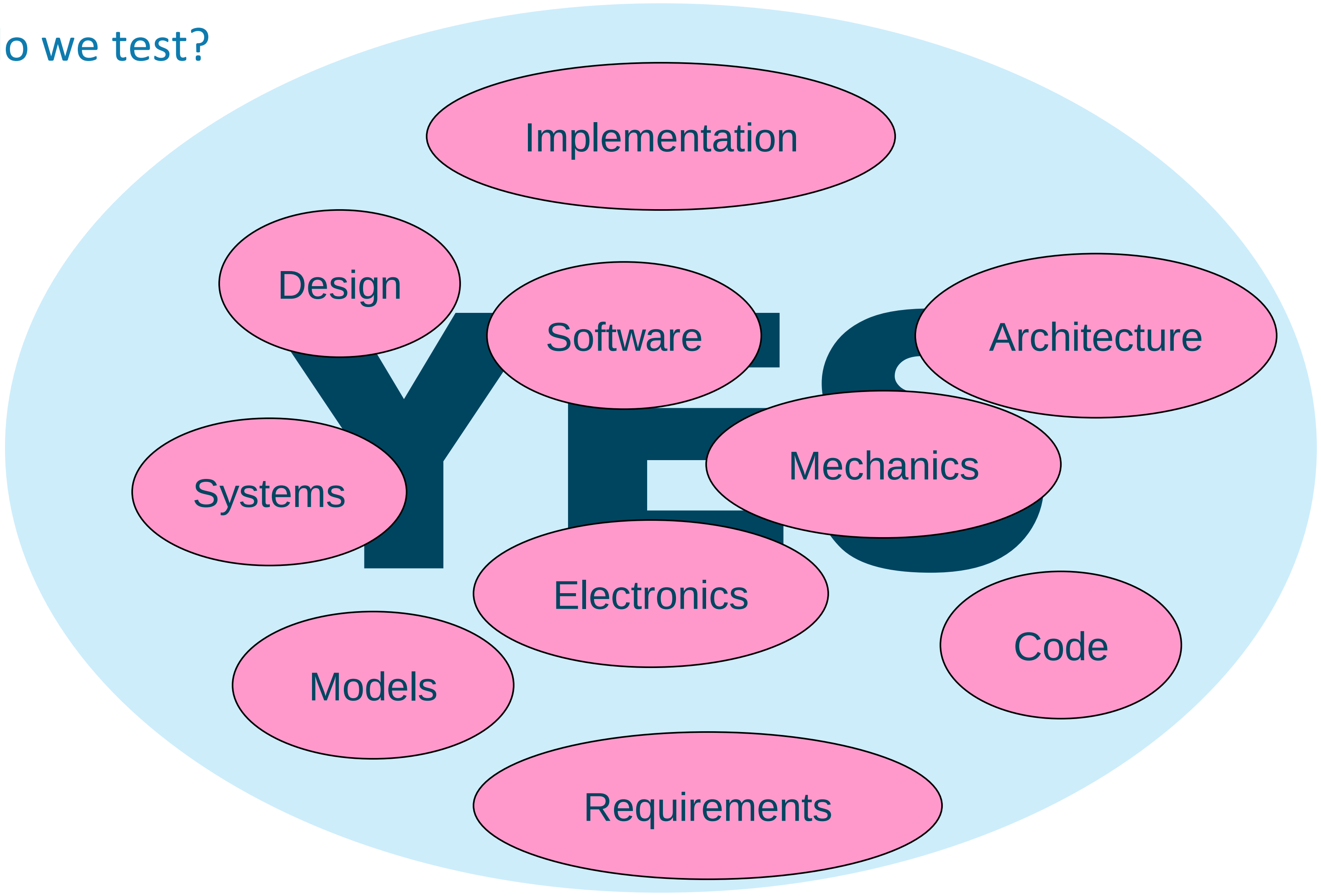


Always works while under attack	Level 4
Always works with invariant violations	Level 3
Always works under normal conditions	Level 2
"Works"	Level 1
Kinda sorta	Level 0

Why do we test?


To uncover limitations and constraints
To find out if it works
To find out if it DOESN'T work
To demonstrate compliance

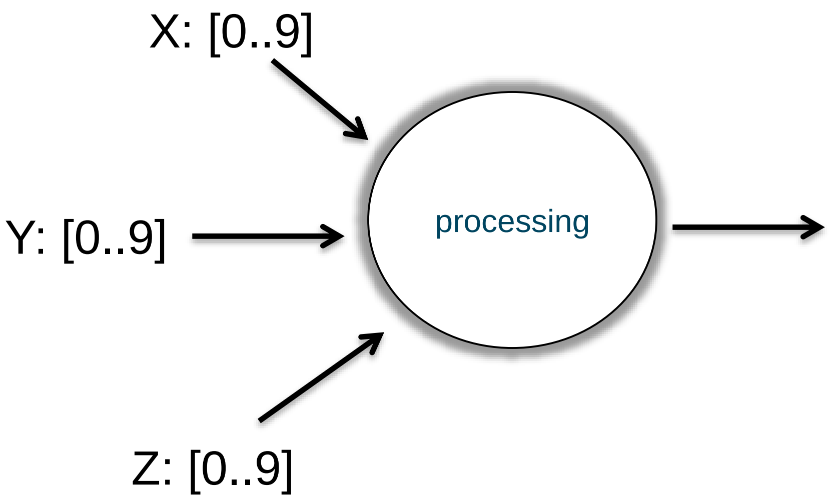
What do we test?



We normally think about testing code *but we can test anything that makes causality assertions and is sufficiently rigorous to be executable*

Why is testing hard?

1. There are (many many) more ways for something to fail than there are for it to succeed
2. Assumptions are often not explicitly stated but their invalidation can cause failures which are both subtle and catastrophic
3. It is both difficult and time consuming to get degrees of test completeness
4. People just as smart as you may be trying to break your system

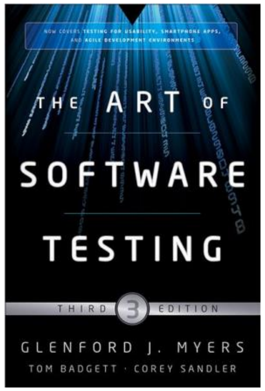


At first look, this has 1000 combinations to be tested. But what if

- X comes before Y? Or Z before X?
- The system expects Z to occur in < 20ms but it arrives at 30ms?
- The output comes too late?
- What if Z, Y, and Z are not independent? Example: if X>5 then Y must be <= 2
- What if X is -1?
- Does the case Z==-20 fail in the same way as X == 45?
- What if X and Y are supplied but not Z?
- Resources (e.g. memory) aren't available for the computation?
- Assumptions (preconditions) are not met?

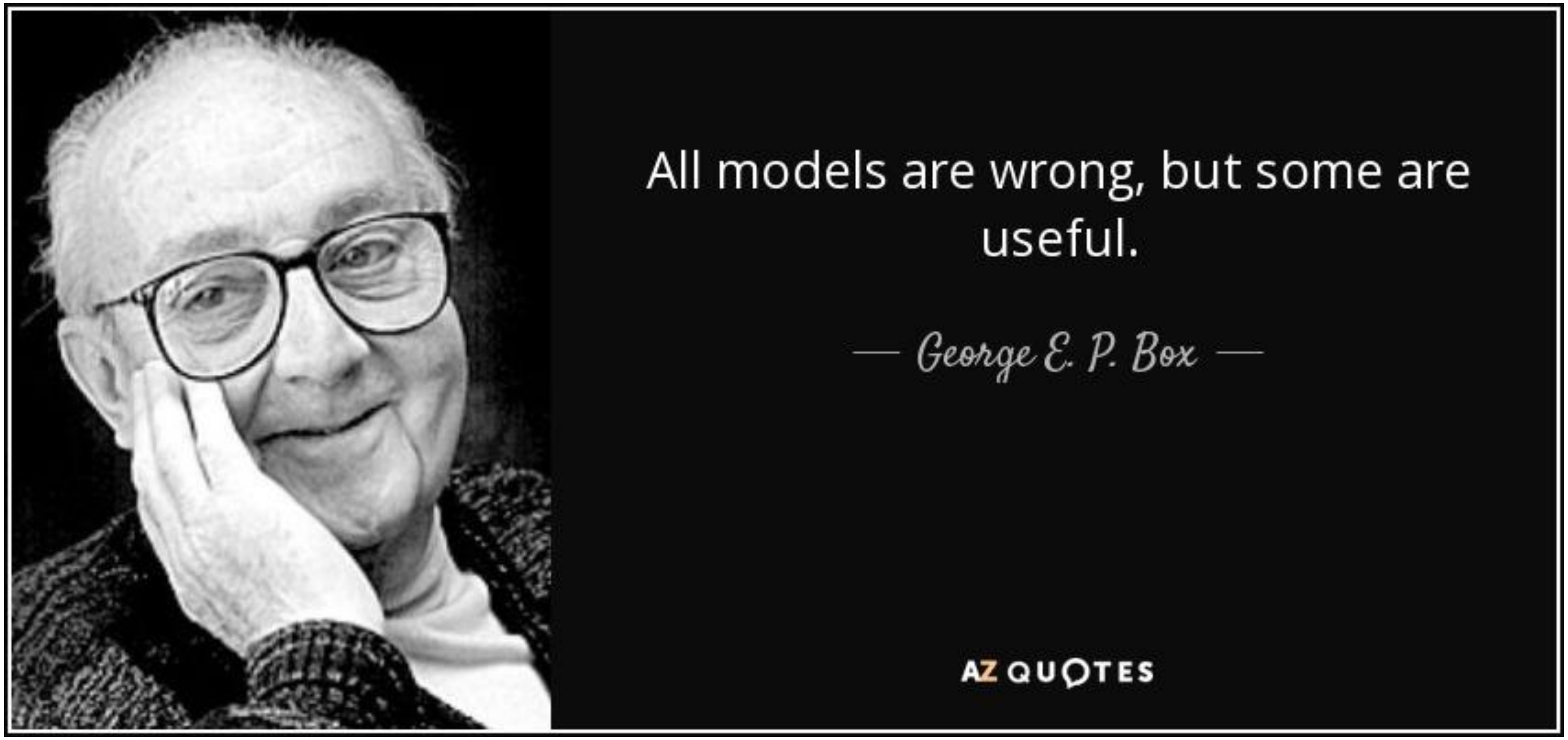
Testing can never be complete – there are an essentially infinite set of combinations of value, sequence, and timing

Glenford Meyer's *The Art of Testing*



- Consider the simple problem
  - *The program reads three integer values from a text input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.*
  - **Define test cases for this system.**
- Did you remember to test
  - Valid scalene triangles? Valid isosceles triangles? Valid equilateral triangles?
  - Have you ensured that it is valid when you swap dimensions on different sides for all types?
  - Did you try an example with a zero length side? Negative number?
  - Did you try specifying the wrong number of sides (e.g. 2 sides or 4 sides)?
  - Did you test the case where the length of one side is the sum of the other two?
  - Did you test with and without whitespace? Alphabetic characters? Special characters?
- Meyer reports highly qualified professional programmers average 7.8 out of 14 tests that he identifies even for this trivial example

Models



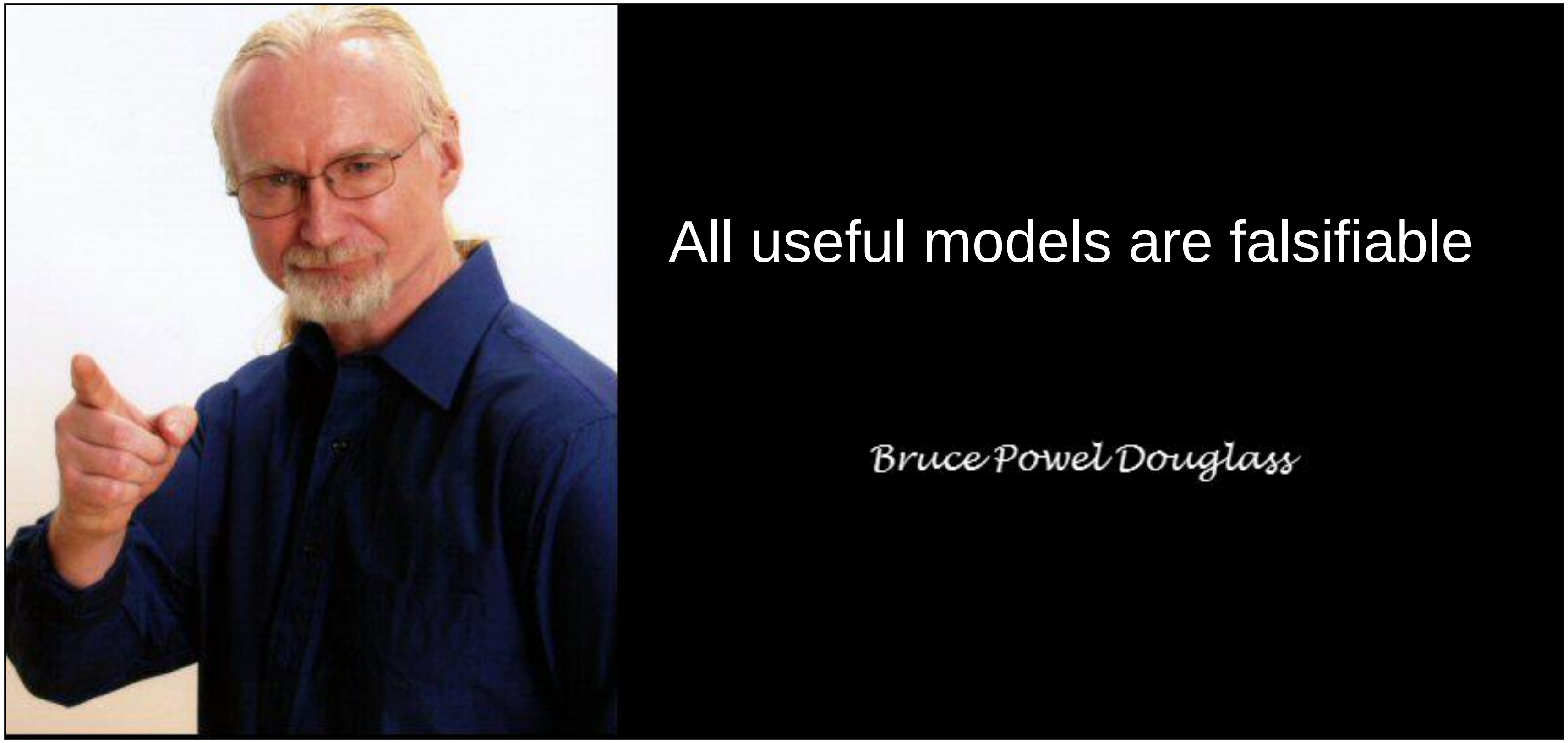
All models are wrong, but some are useful.

— George E. P. Box —

AZ QUOTES

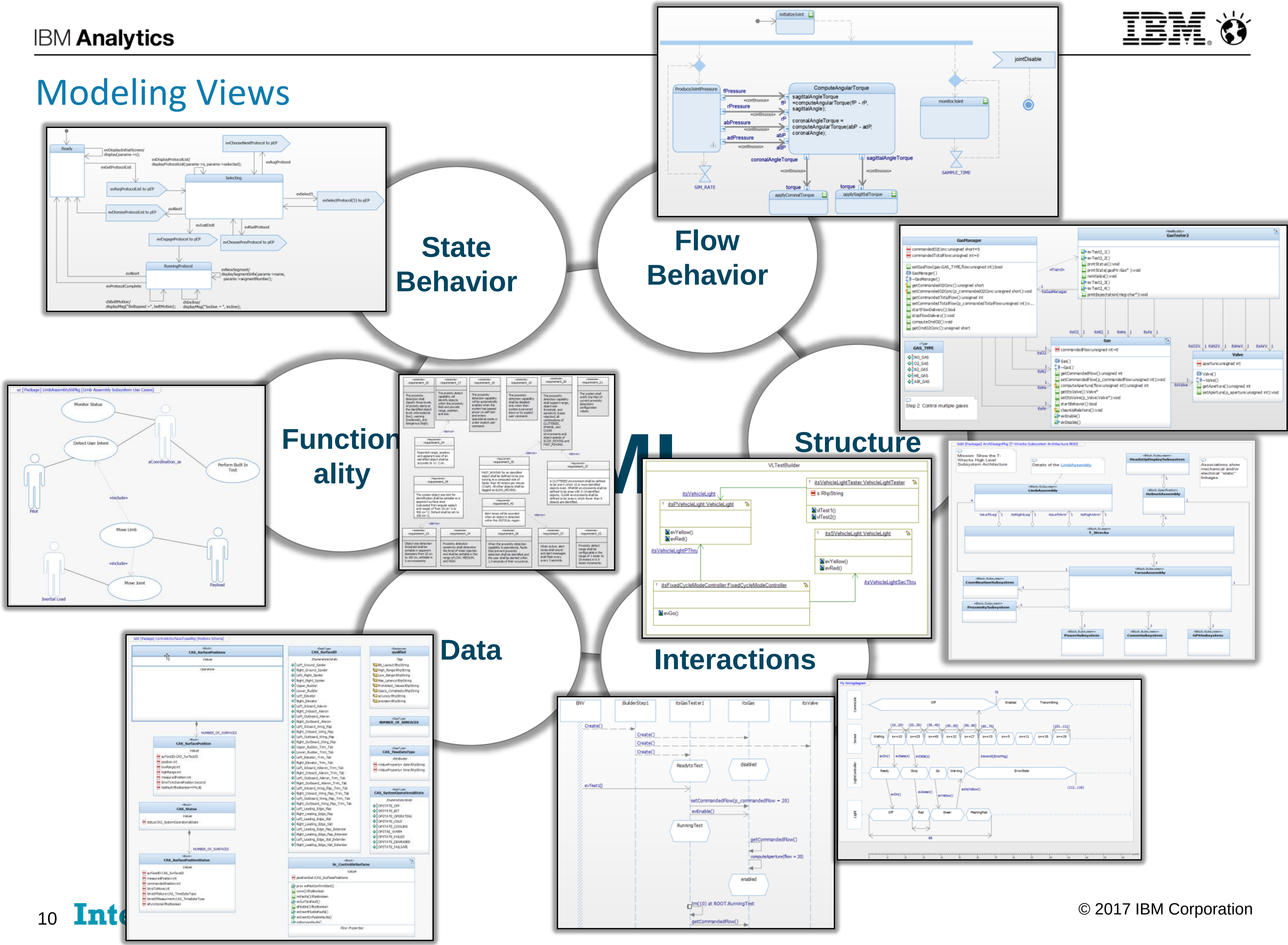
- Problem: Reality is too complex
- Solution: Create a model
- A model is always a simplification of reality, wherein we focus on aspects relevant to things we care about and elide details of those things we do not.

Models



- Rigorously defined – computable – models make statements that can be demonstrated to be true or false
- A subtype of computable models – known as executable models – can be tested

Modeling Views



Kinds of Models

Requirements Models

Requirements Models

Conceptual Models

Implementation Models

Analysis Models

Testing Models

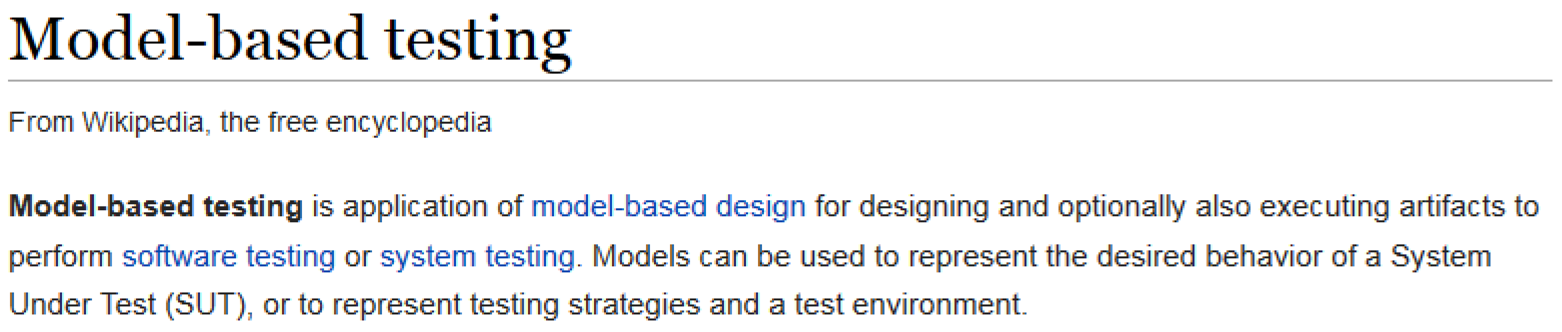
Architecture Models

Design Models

Any of these models can be tested.

It's not just about testing code!

What is model-based testing?



Model-based testing

From Wikipedia, the free encyclopedia

Model-based testing is application of model-based design for designing and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a System Under Test (SUT), or to represent testing strategies and a test environment.

Model-based testing (MBT) means using models…

- to describe test environments
- to describe test strategies
- to generate test cases
- to enable test execution for software and/or system testing
- to implement full traceability between requirements, models, code, and test cases

Automating MBT: What do we want to automate?

- Creation of Test Architecture
- Capturing of outcomes during execution
- Conversion of requirements scenarios to test cases
- Application of test cases to system
- Identification of points of failure
- Gathering of pass/fail statistics
- Computation of coverage metrics

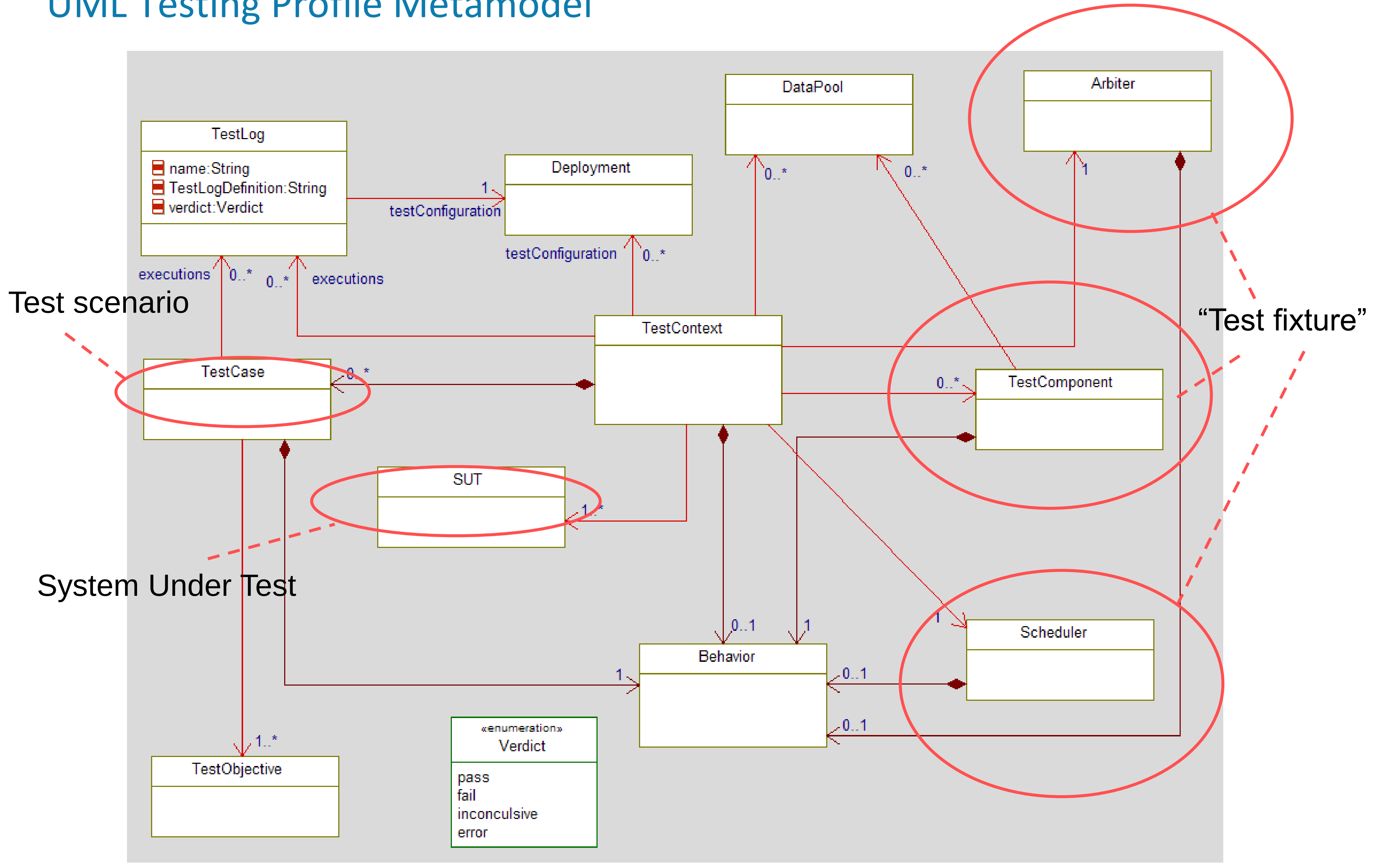
UML Testing Profile

- Current revision 1.2 (April 2013)
  - OMG Document formal/2013-04-03
  - Version 2.0 is in the works
  - Available at http://www.omg.org/spec/UTP/1.2/PDF

The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems. It is a test modeling language that can be used with all major object and component technologies and applied to testing systems in various application domains. The UML Testing Profile can be used stand alone for the handling of test artifacts or in an integrated manner with UML for a handling of system and test artifacts together.

The UML Testing Profile extends UML with test specific concepts like test components, verdicts, defaults, etc. These concepts are grouped into concepts for test architecture, test data, test behavior, and time. Being a profile, the UML testing profile seamlessly integrates into UML: it is based on the UML metamodel and reuses UML syntax. The UML Testing Profile is based on the UML 2.0 specification. The UML Testing Profile is defined by using the metamodeling approach of UML.

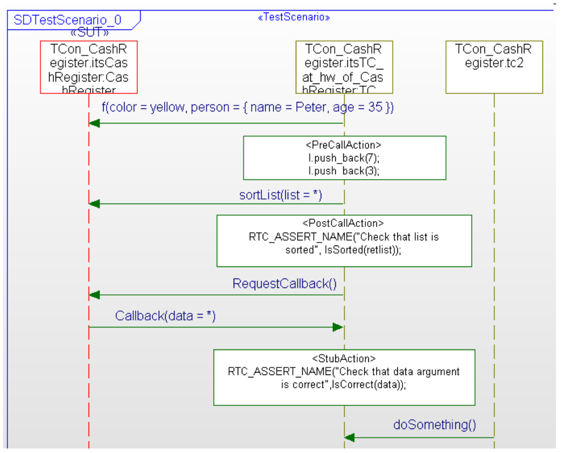
UML Testing Profile Metamodel



InternetofThings

Capture test cases with UML/SysML

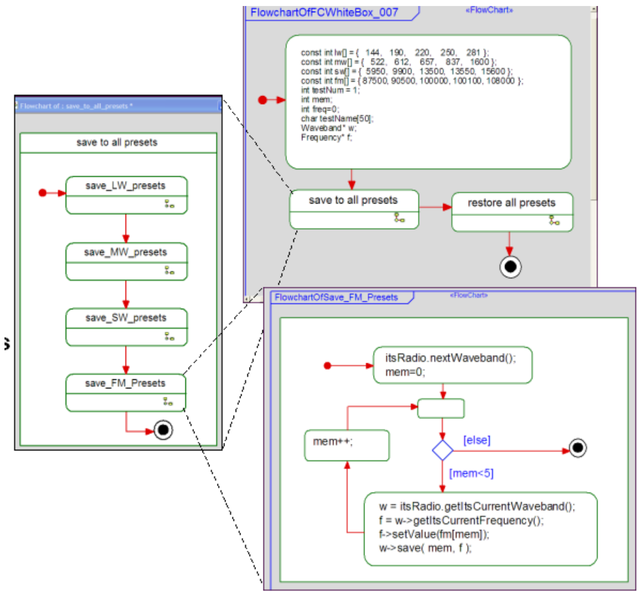
- Recommend using OMG's standard UML Testing Profile (www.omg.org)
- Specify test cases visually for better communication across teams
- Creating code tests cases or importing Cunit/Cpp unit tests also possible
- Can be done manually or with automation (via Test Conductor)


SDTestScenario_0
«TestScenario»
«SUT»
TCon_CashRegister.itsCashRegister:CashRegister
TCon_CashRegister.itsTC_at_hw_of_CashRegisterTC
TCon_CashRegister.tc2
f(color = yellow, person = { name = Peter, age = 35 })
sortList(list = *)
RequestCallback()
Callback(data = *)
doSomething()

```
<PreCallAction>
l.push_back(7);
l.push_back(3);
```

```
<PostCallAction>
RTC_ASSERT_NAME("Check that list is sorted", IsSorted(retlist));
```

```
<StubAction>
RTC_ASSERT_NAME("Check that data argument is correct",IsCorrect(data));
```

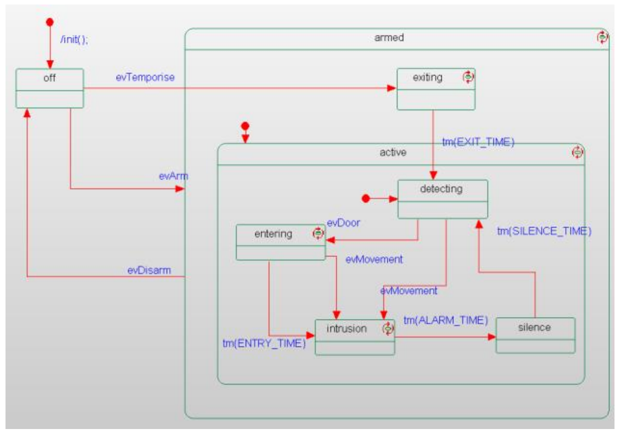


```
const int lw[] = {   144,   190,   220,   250,   281 };
const int mw[] = {   522,   612,   657,   837,  1600 };
const int sw[] = {  5950,  9900, 13500, 13550, 15600 };
const int fm[] = { 87500, 90500, 100000, 100100, 108000 };
int testNum = 1;
int mem;
int freq=0;
char testName[50];
Waveband* w;
Frequency* f;
```

```
itsRadio.nextWaveband();
mem=0;
```

```
mem++;
```

```
w = itsRadio.getItsCurrentWaveband();
f = w->getItsCurrentFrequency();
f->setValue(fm[mem]);
w->save( mem, f );
```



Sequence Diagram Test Case

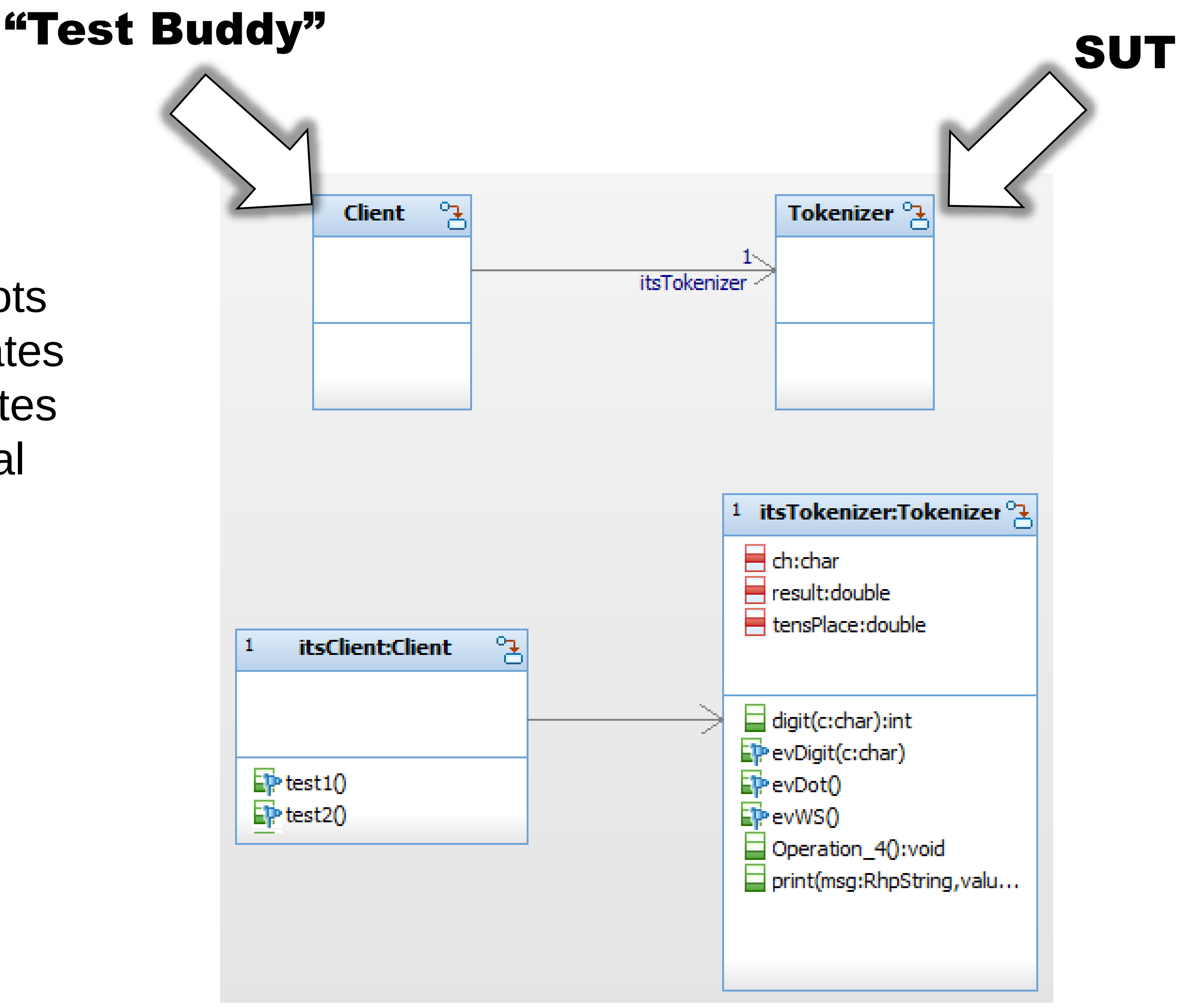
Flow Chart Test Cases

Statechart Test Case

Internet of Things

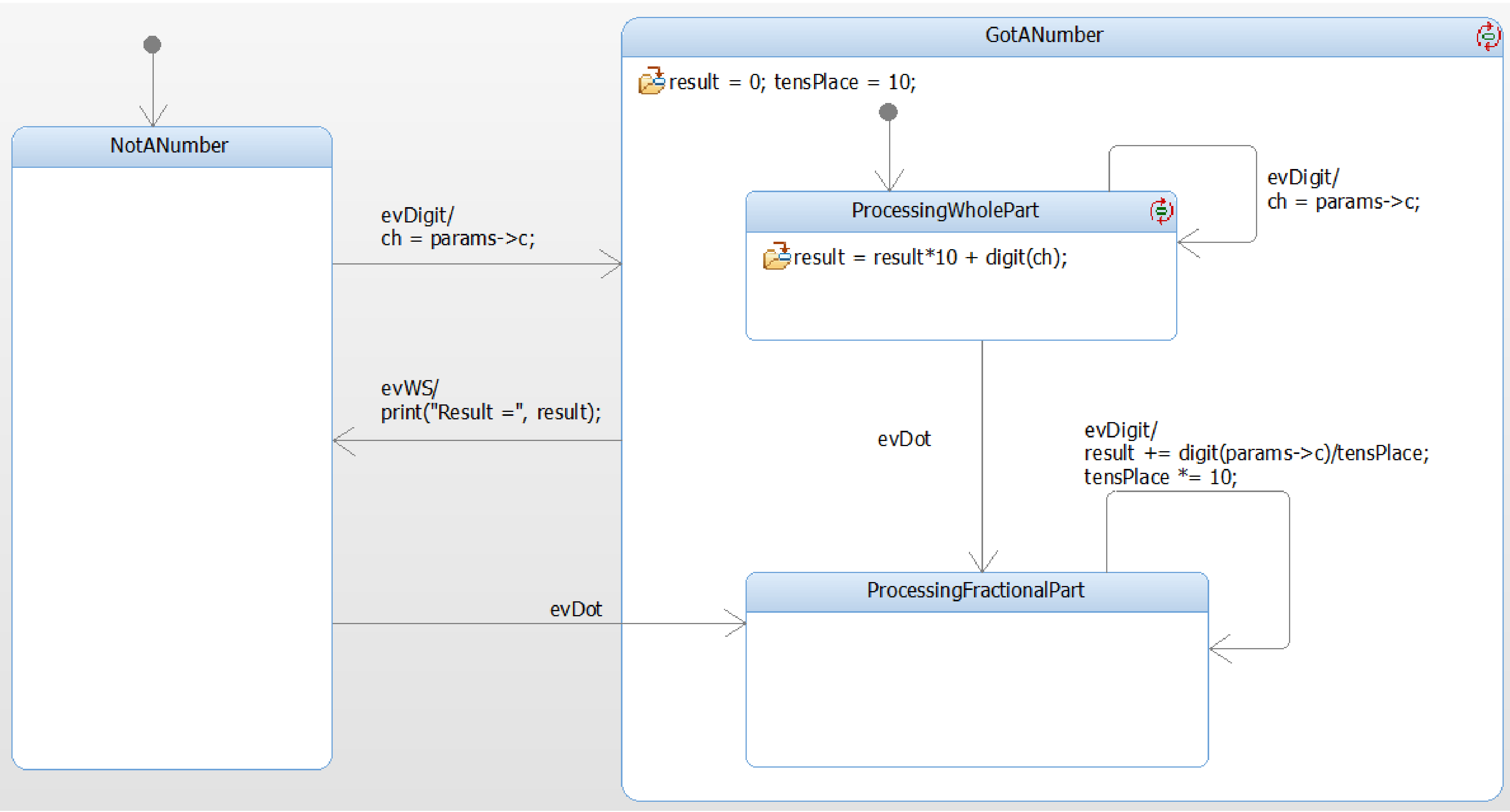
Example model: Tokenizer (Manual)

This simple model receives digits and dots as characters, evaluates the string and computes the corresponding real value



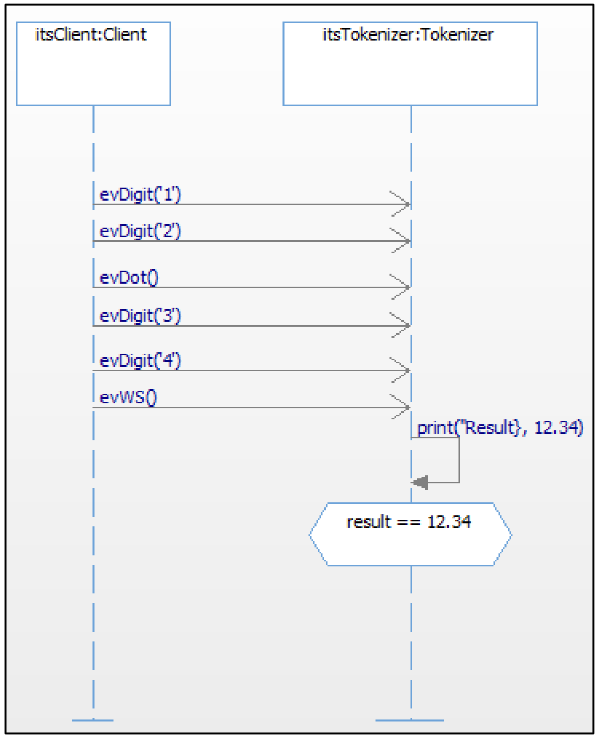
Example model: Tokenizer (Manual)

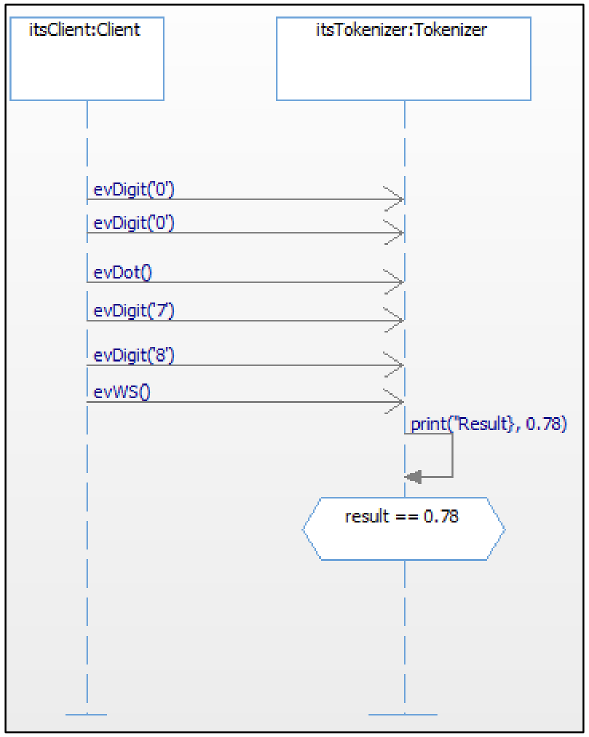
This is the state machine for the Tokenizer class

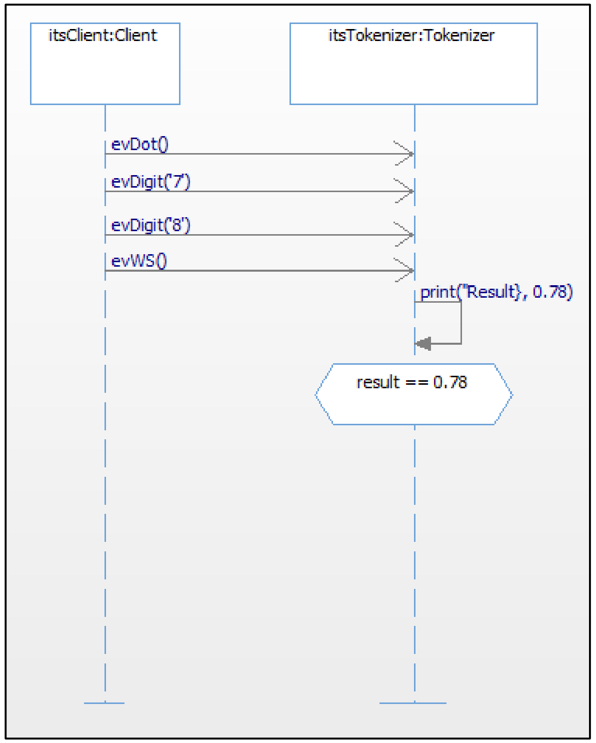


Example model: Tokenizer (Manual)

Create Test Cases as Sequence Diagrams

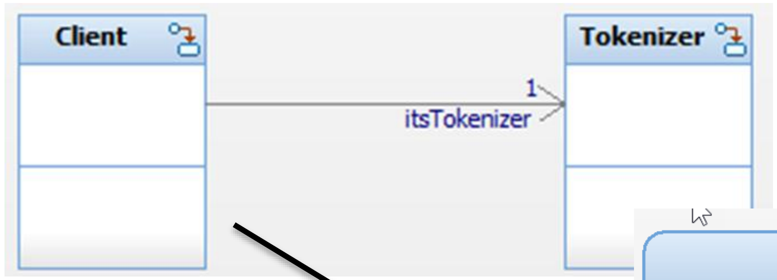


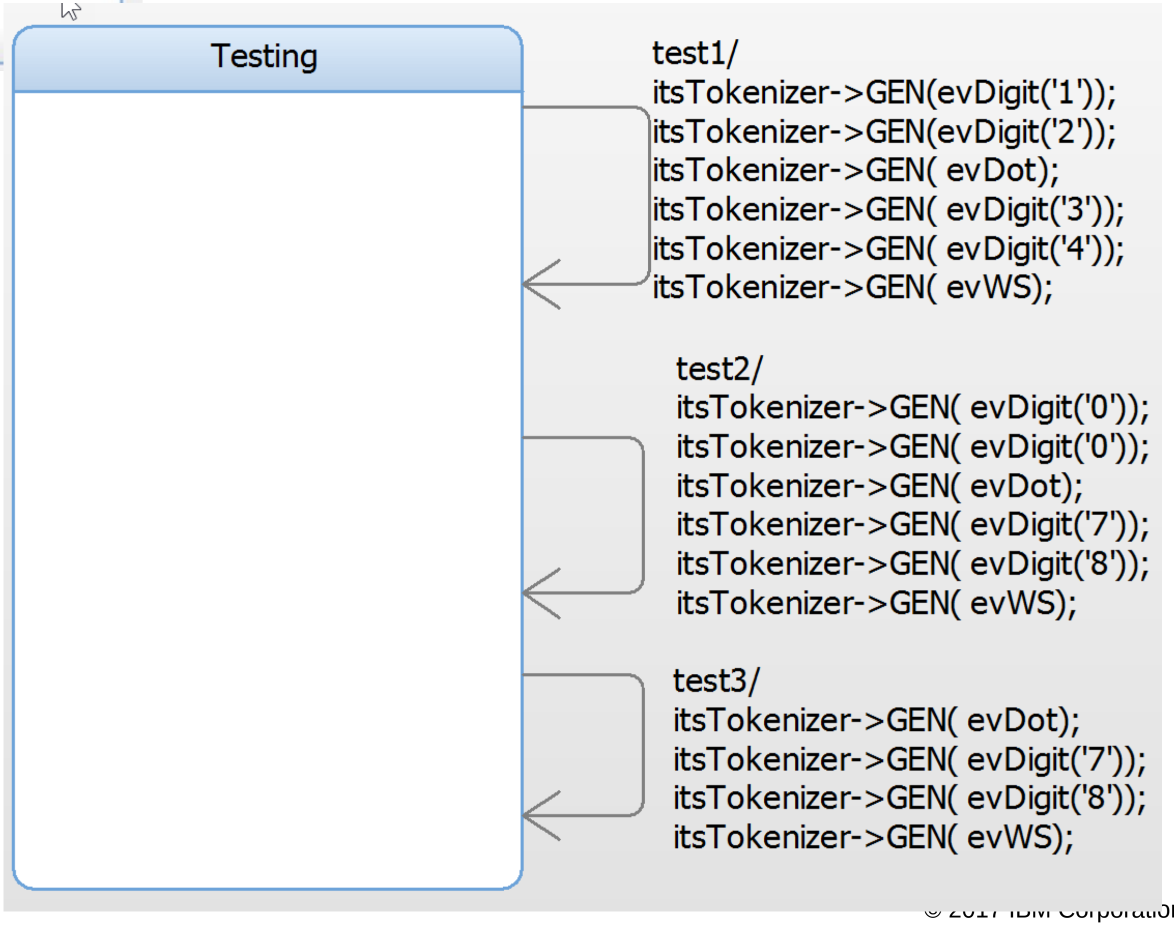




Example model: Tokenizer (Manual)

Manually instrument the client (Test Buddy) to invoke the test





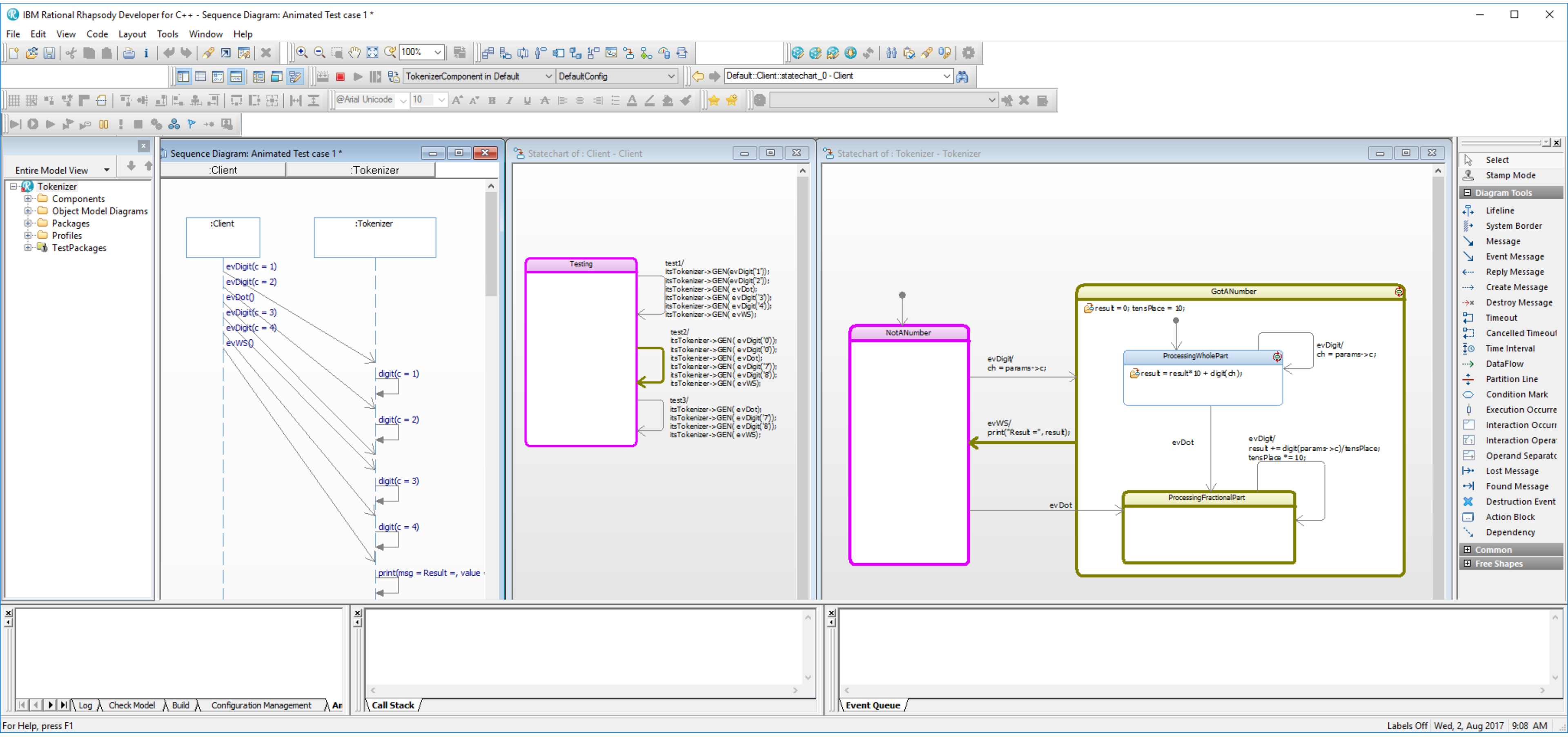
```
test1/
itsTokenizer->GEN(evDigit('1'));
itsTokenizer->GEN(evDigit('2'));
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('3'));
itsTokenizer->GEN( evDigit('4'));
itsTokenizer->GEN( evWS);
```

```
test2/
itsTokenizer->GEN( evDigit('0'));
itsTokenizer->GEN( evDigit('0'));
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('7'));
itsTokenizer->GEN( evDigit('8'));
itsTokenizer->GEN( evWS);
```

```
test3/
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('7'));
itsTokenizer->GEN( evDigit('8'));
itsTokenizer->GEN( evWS);
```

Example model: Tokenizer (Manual)

Now execute the model and create “animated sequence diagrams”* from the execution)



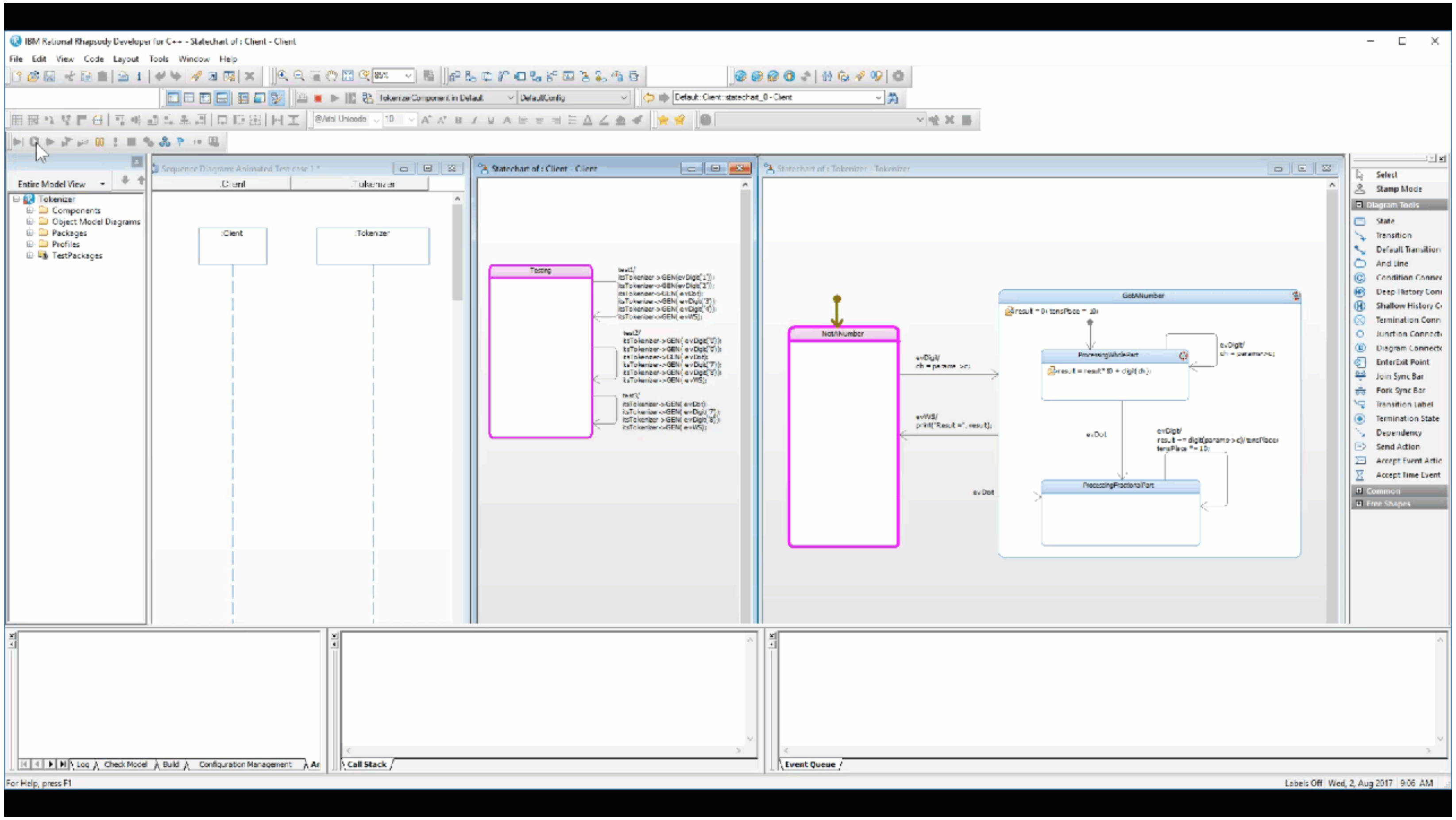
```
test1/
itsTokenizer->GEN(evDigit('1'));
itsTokenizer->GEN(evDigit('2'));
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('3'));
itsTokenizer->GEN( evDigit('4'));
itsTokenizer->GEN( evWS);

test2/
itsTokenizer->GEN( evDigit('0'));
itsTokenizer->GEN( evDigit('0'));
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('7'));
itsTokenizer->GEN( evDigit('8'));
itsTokenizer->GEN( evWS);

test3/
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('7'));
itsTokenizer->GEN( evDigit('8'));
itsTokenizer->GEN( evWS);
```

```
result = 0; tensPlace = 10;
result = result* 10 + digit( ch );

evDigit/
ch = params->c;

evWS/
print("Result =", result);

evDigit/
ch = params->c;

evDigit/
result += digit(params->c)/tensPlace;
tensPlace *= 10;
```

* Rhapsody feature – can produce sequence diagrams from the interaction of modelled elements during execution

Example model: Tokenizer (Manual)

Now execute the model and create “animated sequence diagrams”* from the execution)



```
test1/
itsTokenizer->GEN(evDigit('1'));
itsTokenizer->GEN(evDigit('2'));
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('3'));
itsTokenizer->GEN( evDigit('4'));
itsTokenizer->GEN( evWS);
```

```
test2/
itsTokenizer->GEN( evDigit('0'));
itsTokenizer->GEN( evDigit('0'));
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('7'));
itsTokenizer->GEN( evDigit('8'));
itsTokenizer->GEN( evWS);
```

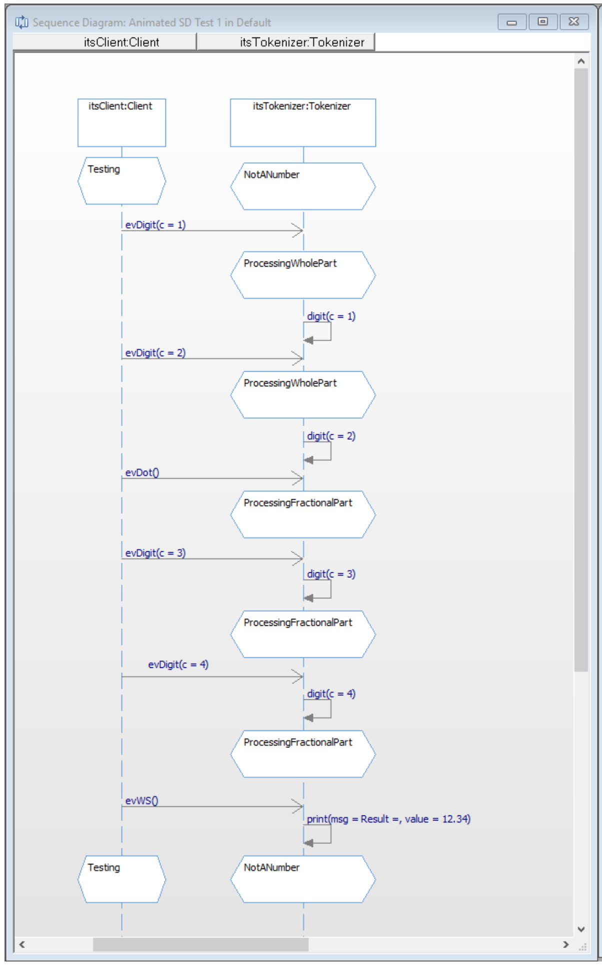
```
test3/
itsTokenizer->GEN( evDot);
itsTokenizer->GEN( evDigit('7'));
itsTokenizer->GEN( evDigit('8'));
itsTokenizer->GEN( evWS);
```

```
result = 0; tensPlace = 10;
result = result * 10 + digit(ch);
evDigit/
ch = params->c;
evWS/
printf("Result =", result);
evDigit/
result += digit(params->c)/tensPlace;
tensPlace *= 10;
```

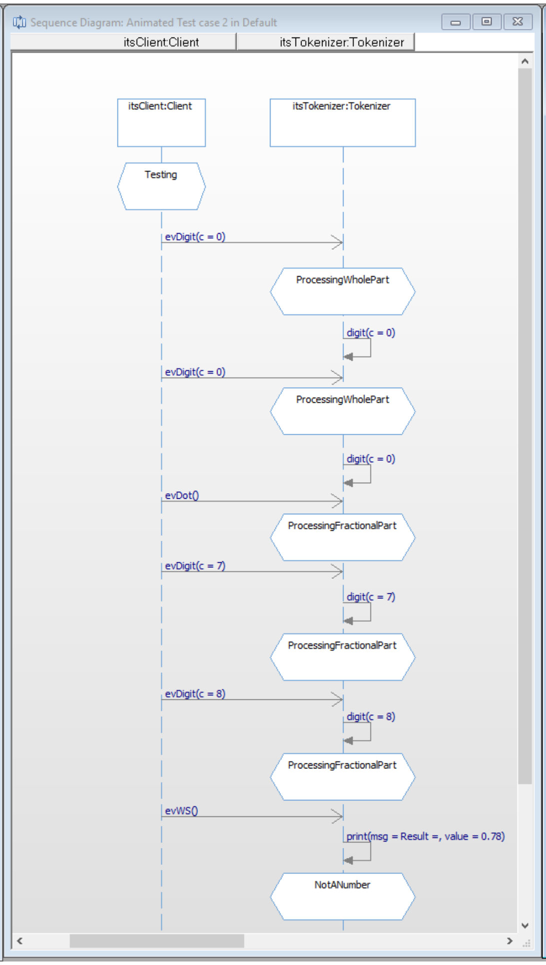
* Rhapsody feature – can produce sequence diagrams from the interaction of modelled elements during execution

Example model: Tokenizer (Manual)

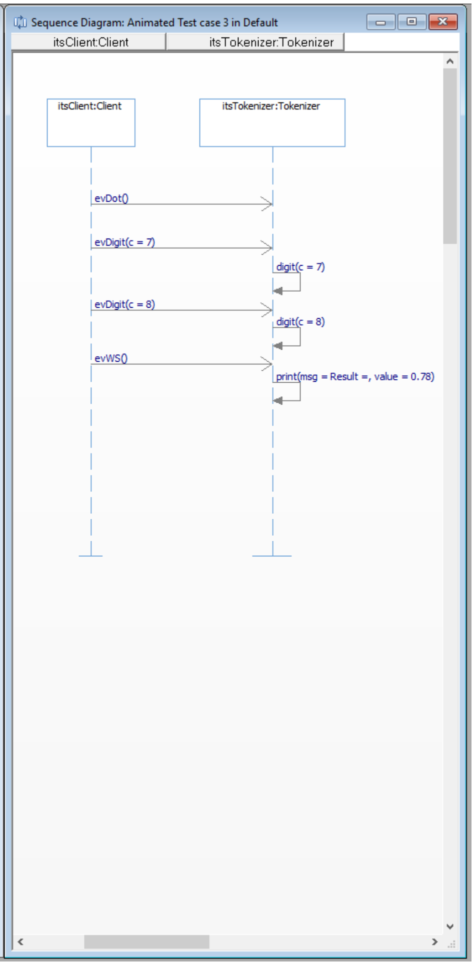
Review the outcomes and compare to the test specifications



Test Case 1 Outcome



Test Case 2 Outcome



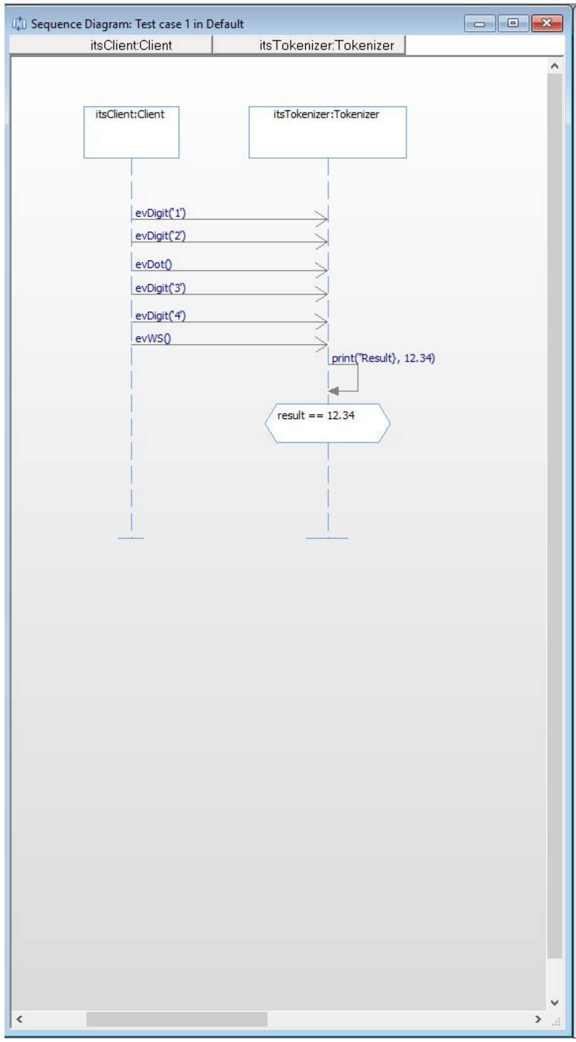
Test Case 3 Outcome

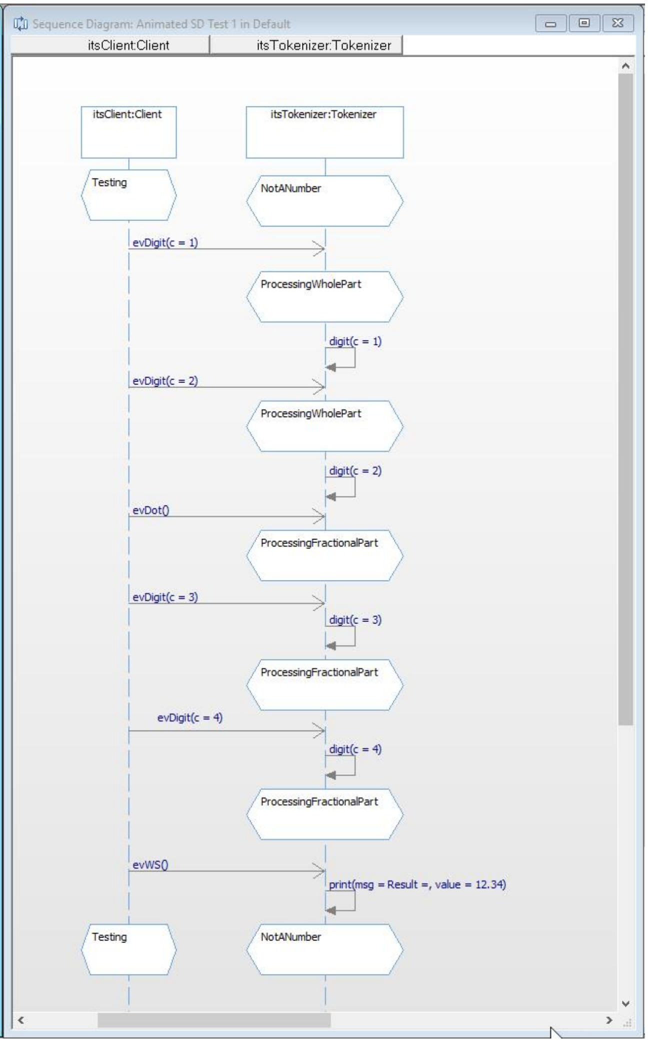
Internet of Things

Example model: Tokenizer (Manual)

Review the outcomes and compare to the test specifications

Test Case 1

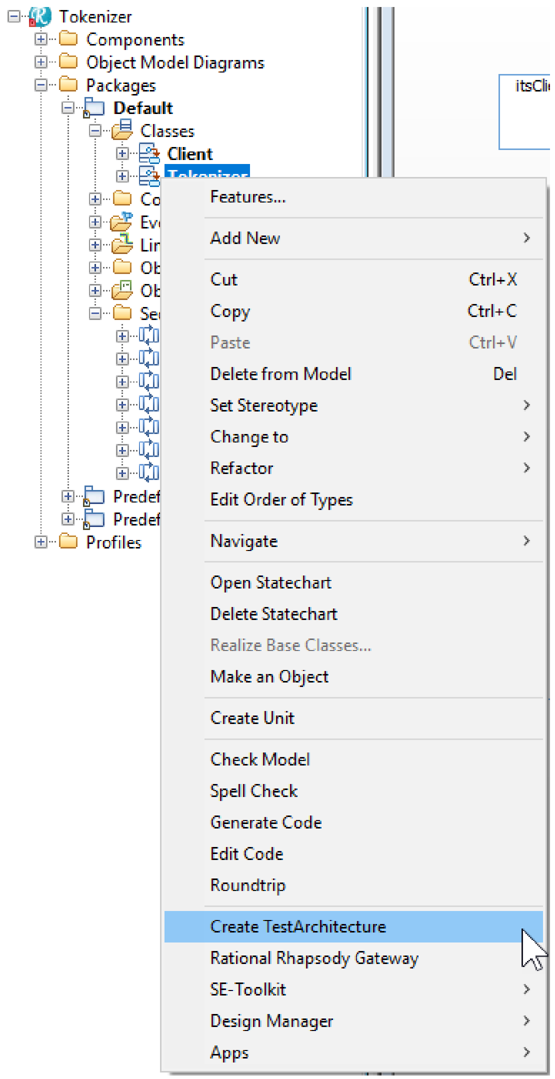




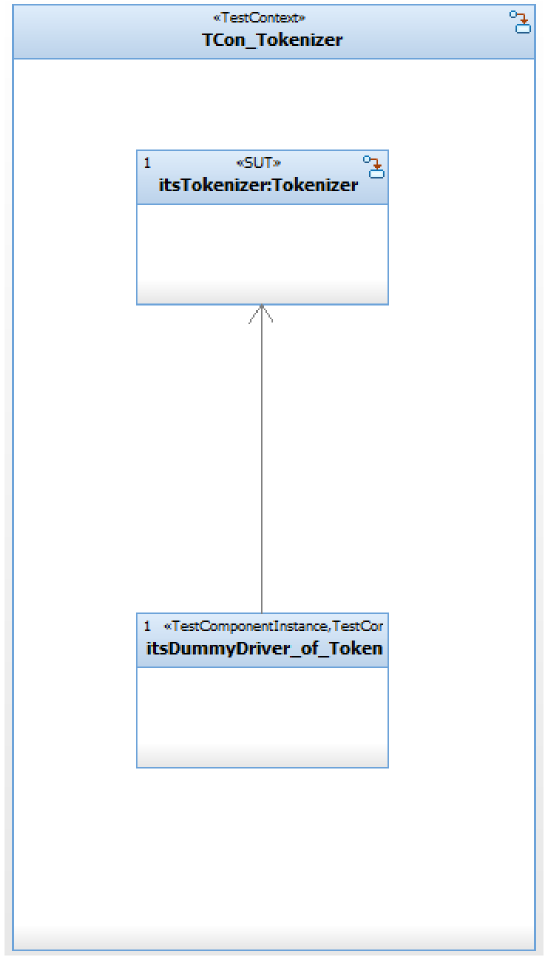
Test Case 1 Result

Internet of Things

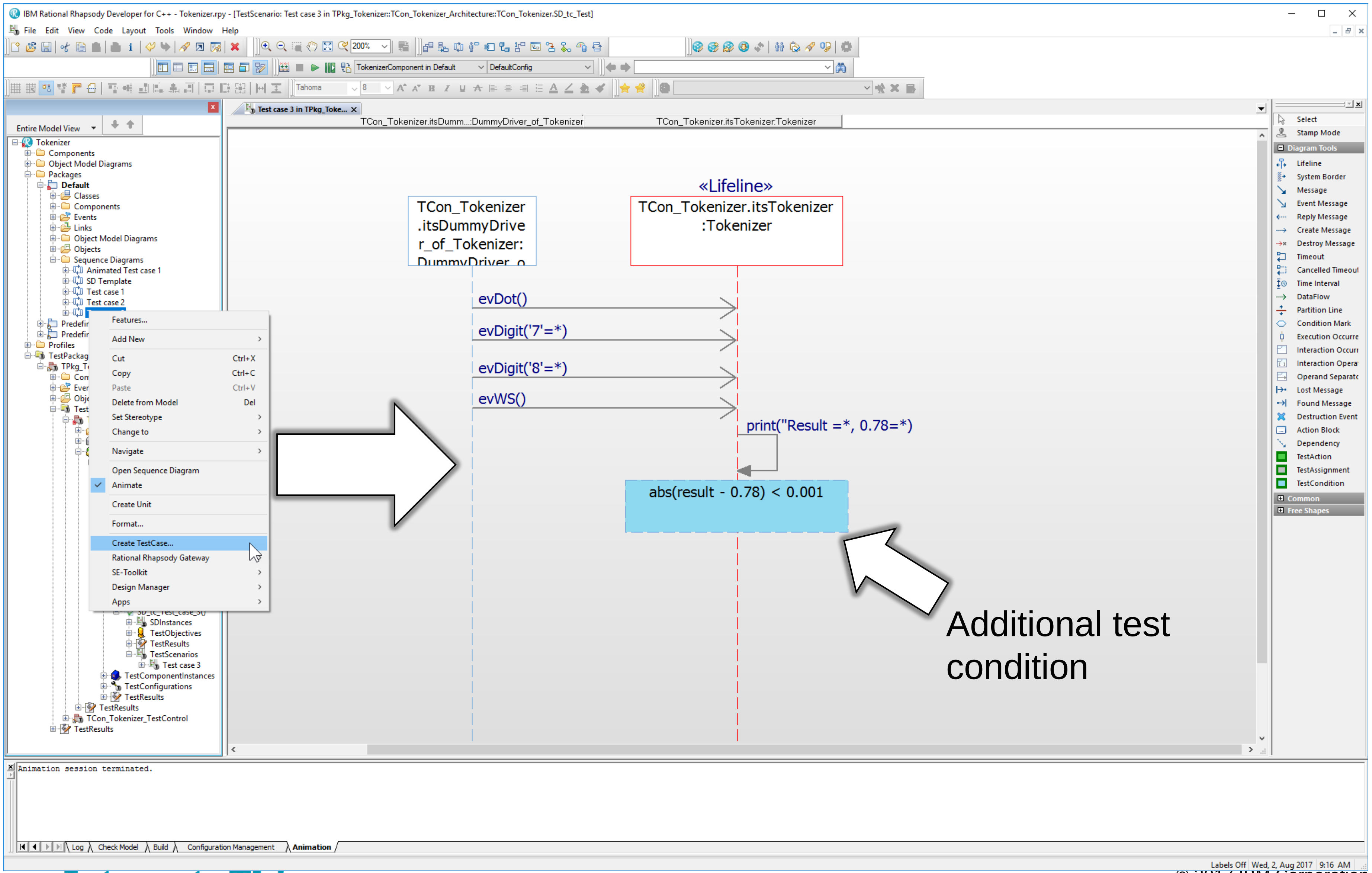
Example Model: Tokenizer (Test Conductor)



Generates



Example Model: Tokenizer (Test Conductor)



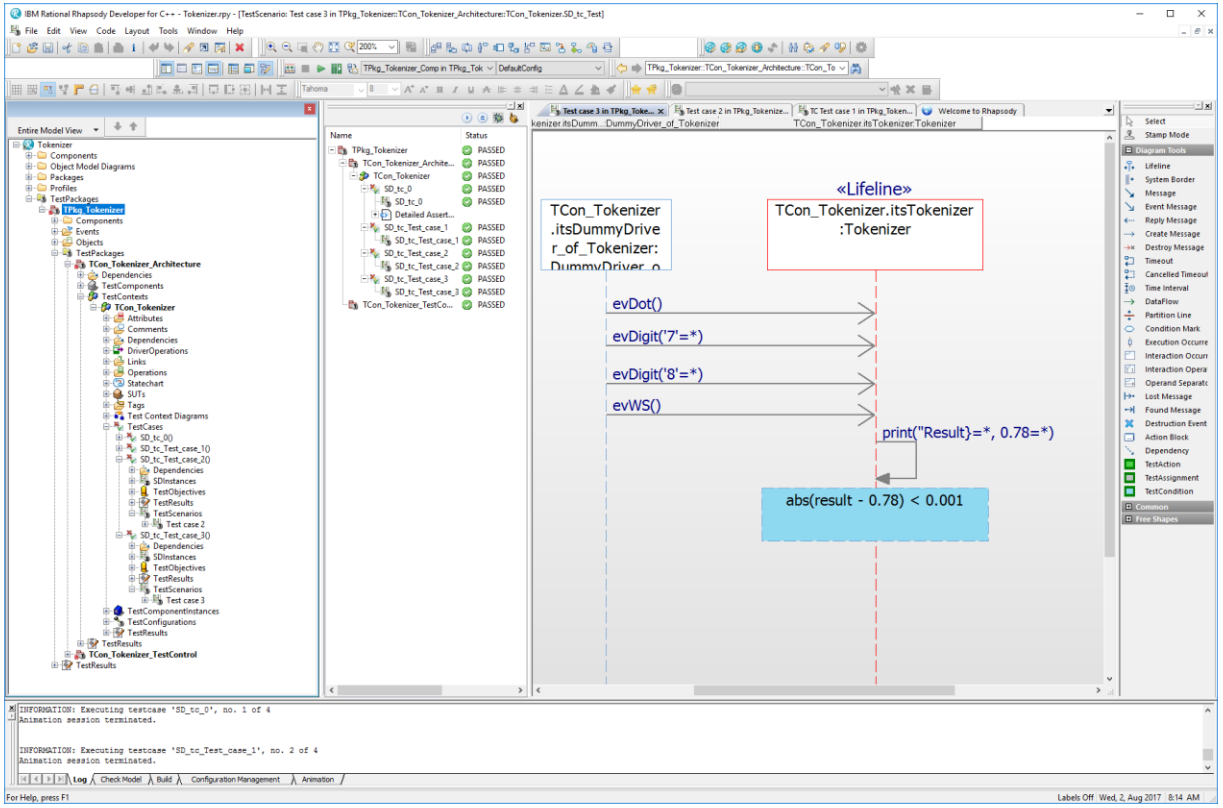
```
Animation session terminated.
```

Additional test condition

InternetofThings

Example Model: Tokenizer (Test Conductor)

Test outcomes



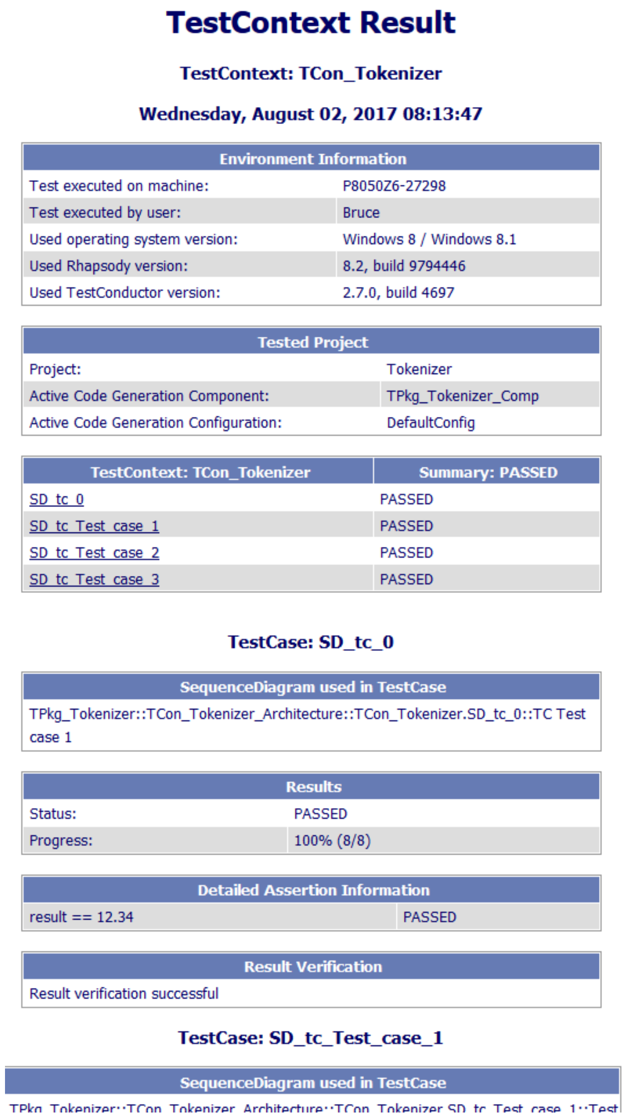
IBM Rational Rhapsody Developer for C++ - Tokenizer.rpy - [TestScenario: Test case 3 in TPkg_Tokenizer::TCon_Tokenizer_Architecture::TCon_Tokenizer.SD_tc_Test]

Name	Status
TPkg_Tokenizer	PASSED
TCon_Tokenizer_Archite...	PASSED
TCon_Tokenizer	PASSED
SD_tc_0	PASSED
SD_tc_0	PASSED
Detailed Assert...	
SD_tc_Test_case_1	PASSED
SD_tc_Test_case_1	PASSED
SD_tc_Test_case_2	PASSED
SD_tc_Test_case_2	PASSED
SD_tc_Test_case_3	PASSED
SD_tc_Test_case_3	PASSED
TCon_Tokenizer_TestCo...	PASSED

TCon_Tokenizer .itsDummyDriver_of_Tokenizer: DummyDriver_o
«Lifeline»
TCon_Tokenizer.itsTokenizer :Tokenizer
evDot()
evDigit('7'=*)
evDigit('8'=*)
evWS()
print("Result}=*, 0.78=*)
abs(result - 0.78) < 0.001

```
INFORMATION: Executing testcase 'SD_tc_0', no. 1 of 4
Animation session terminated.


INFORMATION: Executing testcase 'SD_tc_Test_case_1', no. 2 of 4
Animation session terminated.
```

Log
Check Model
Build
Configuration Management
Animation
For Help, press F1



TestContext Result

TestContext: TCon_Tokenizer

Wednesday, August 02, 2017 08:13:47

Environment Information	
Test executed on machine:	P8050Z6-27298
Test executed by user:	Bruce
Used operating system version:	Windows 8 / Windows 8.1
Used Rhapsody version:	8.2, build 9794446
Used TestConductor version:	2.7.0, build 4697

Tested Project	
Project:	Tokenizer
Active Code Generation Component:	TPkg_Tokenizer_Comp
Active Code Generation Configuration:	DefaultConfig

TestContext: TCon_Tokenizer	Summary: PASSED
SD_tc_0	PASSED
SD_tc_Test_case_1	PASSED
SD_tc_Test_case_2	PASSED
SD_tc_Test_case_3	PASSED

TestCase: SD_tc_0

SequenceDiagram used in TestCase
TPkg_Tokenizer::TCon_Tokenizer_Architecture::TCon_Tokenizer.SD_tc_0::TC Test case 1

Results	
Status:	PASSED
Progress:	100% (8/8)

Detailed Assertion Information	
result == 12.34	PASSED

Result Verification
Result verification successful

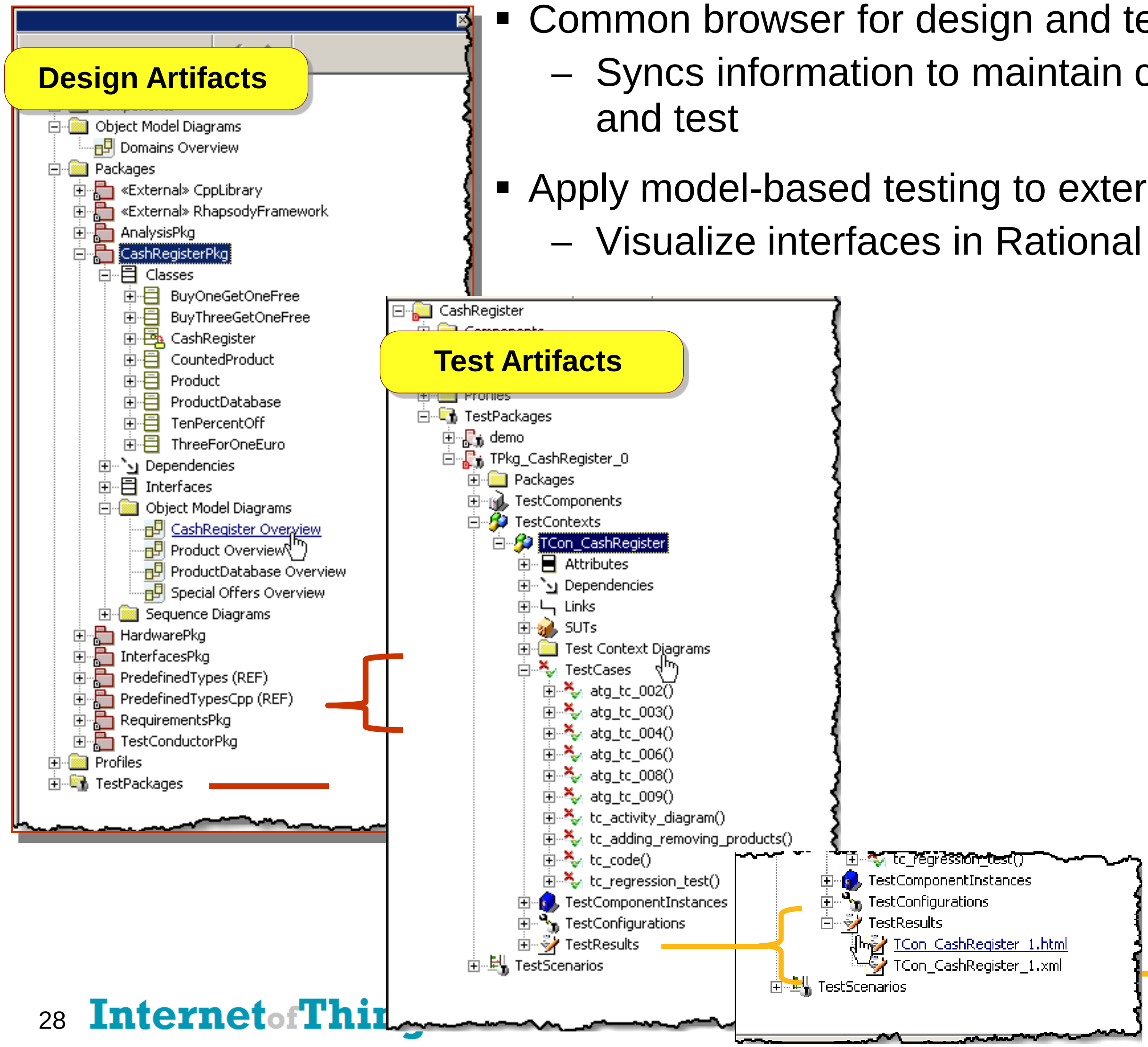
TestCase: SD_tc_Test_case_1

SequenceDiagram used in TestCase
TPkg_Tokenizer::TCon_Tokenizer_Architecture::TCon_Tokenizer.SD_tc_Test_case_1::Test

Test Report

Integrated design and test environment with automation
Manage test cases within Rational Rhapsody with Test Conductor

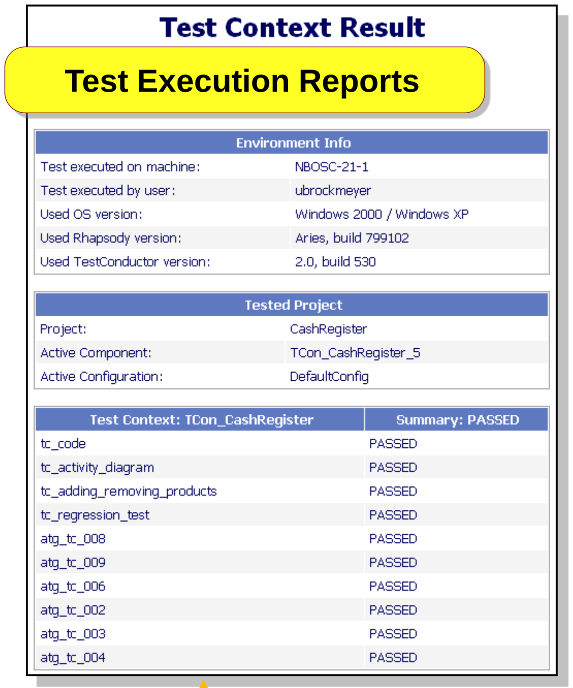
- Common browser for design and test information
  - Syncs information to maintain consistency between design and test
- Apply model-based testing to external code
  - Visualize interfaces in Rational Rhapsody



Object Model Diagrams
Domains Overview
Packages
«External» CppLibrary
«External» RhapsodyFramework
AnalysisPkg
CashRegisterPkg
Classes
BuyOneGetOneFree
BuyThreeGetOneFree
CashRegister
CountedProduct
Product
ProductDatabase
TenPercentOff
ThreeForOneEuro
Dependencies
Interfaces
Object Model Diagrams
CashRegister Overview
Product Overview
ProductDatabase Overview
Special Offers Overview
Sequence Diagrams
HardwarePkg
InterfacesPkg
PredefinedTypes (REF)
PredefinedTypesCpp (REF)
RequirementsPkg
TestConductorPkg
Profiles
TestPackages

CashRegister
Profiles
TestPackages
demo
TPkg_CashRegister_0
Packages
TestComponents
TestContexts
TCon_CashRegister
Attributes
Dependencies
Links
SUTs
Test Context Diagrams
TestCases
atg_tc_002()
atg_tc_003()
atg_tc_004()
atg_tc_006()
atg_tc_008()
atg_tc_009()
tc_activity_diagram()
tc_adding_removing_products()
tc_code()
tc_regression_test()
TestComponentInstances
TestConfigurations
TestResults
TestScenarios

tc_regression_test()
TestComponentInstances
TestConfigurations
TestResults
TCon_CashRegister_1.html
TCon_CashRegister_1.xml
TestScenarios



Test Context Result

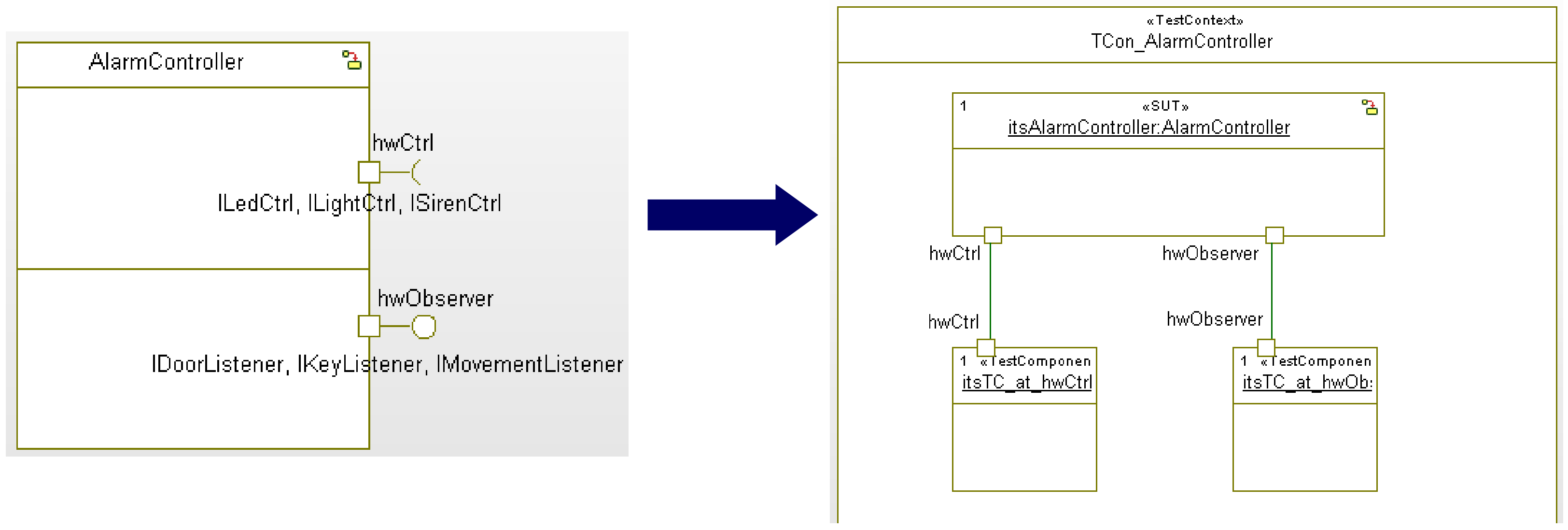
Environment Info	
Test executed on machine:	NBOSC-21-1
Test executed by user:	ubrockmeyer
Used OS version:	Windows 2000 / Windows XP
Used Rhapsody version:	Aries, build 799102
Used TestConductor version:	2.0, build 530

Tested Project	
Project:	CashRegister
Active Component:	TCon_CashRegister_5
Active Configuration:	DefaultConfig

Test Context: TCon_CashRegister	Summary: PASSED
tc_code	PASSED
tc_activity_diagram	PASSED
tc_adding_removing_products	PASSED
tc_regression_test	PASSED
atg_tc_008	PASSED
atg_tc_009	PASSED
atg_tc_006	PASSED
atg_tc_002	PASSED
atg_tc_003	PASSED
atg_tc_004	PASSED

Automate quality

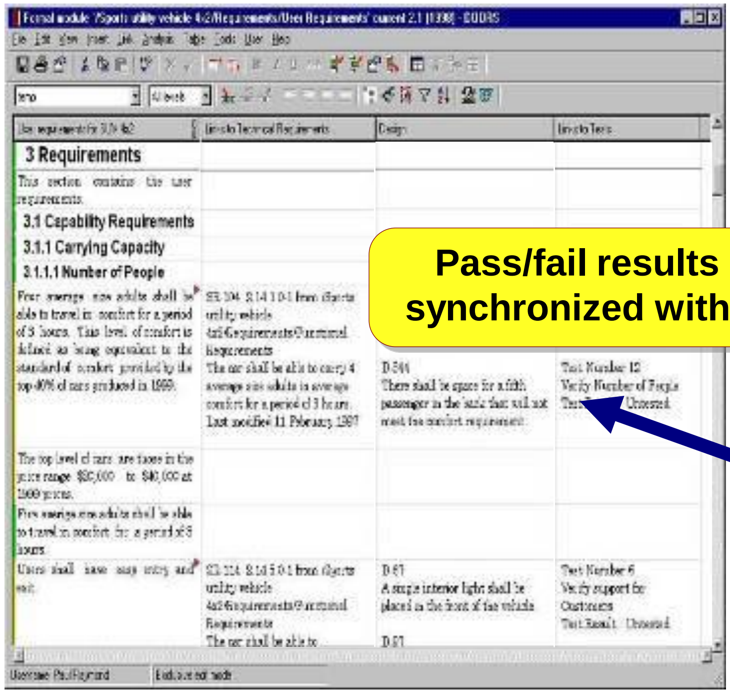
- Automatically create test architecture
  - Creates a System Under Test (SUT), test components and test context
- Apply model-based testing to external code
  - Code is developed outside of Rational Rhapsody
  - Visualize code interfaces in Rational Rhapsody and apply model-based testing



Automatically Created Test Architecture

Requirements-driven testing

- Quick definition and execution of model and requirement-aware tests
  - Unit, integration and system testing
  - Reuse design scenarios as test cases
- Requirement change impact and analysis
  - Know which part of the model or which tests are affected by changing requirements

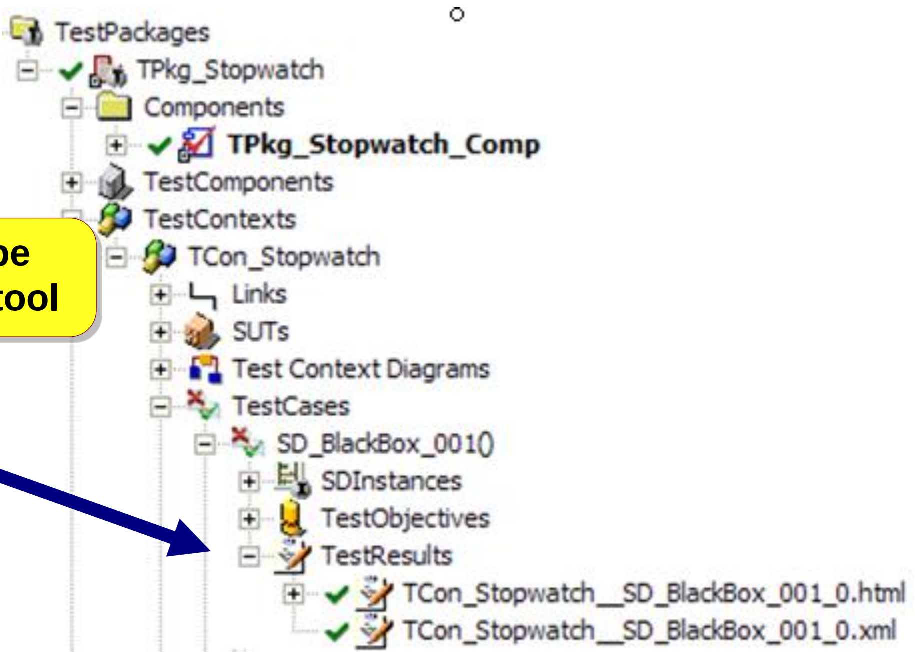


Formal module '/Sports utility vehicle 4x2/Requirements/User Requirements' current 2.1 (1998) - DOORS

User requirements for SUV 4x2	Links to Technical Requirements	Design	Links to Tests
3 Requirements			
This section contains the user requirements.			
3.1 Capability Requirements			
3.1.1 Carrying Capacity			
3.1.1.1 Number of People			
Four average size adults shall be able to travel in comfort for a period of 3 hours. This level of comfort is defined as being equivalent to the standard of comfort provided by the top 40% of cars produced in 1999.	SR 104 8.14.1.0-1 from /Sports utility vehicle 4x2/Requirements/Functional Requirements The car shall be able to carry 4 average size adults in average comfort for a period of 3 hours. Last modified 11 February 1997	D 544 There shall be space for a fifth passenger in the back that will not meet the comfort requirement.	Test Number 12 Verify Number of People Test Result Untested
The top level of cars are those in the price range $30,000 to $40,000 at 1999 prices.			
Five average size adults shall be able to travel in comfort for a period of 3 hours.			
Users shall have easy entry and exit.	SR 114 8.14.5.0-1 from /Sports utility vehicle 4x2/Requirements/Functional Requirements The car shall be able to	D 57 A single interior light shall be placed in the front of the vehicle D 57	Test Number 6 Verify support for Customers Test Result Untested

Username Paul Raynard | Exclusive edit mode

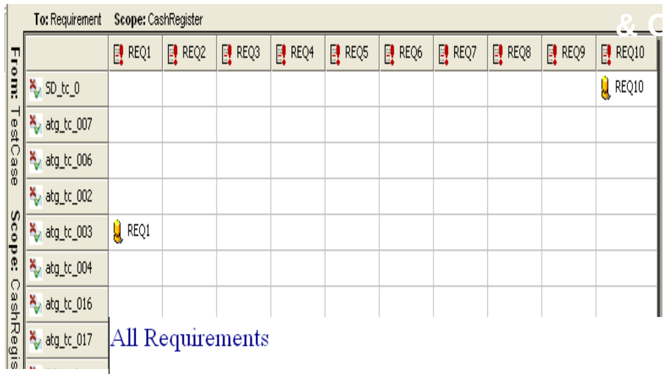
Pass/fail results can be synchronized with RM tool



Internet of Things

Requirements to test results coverage

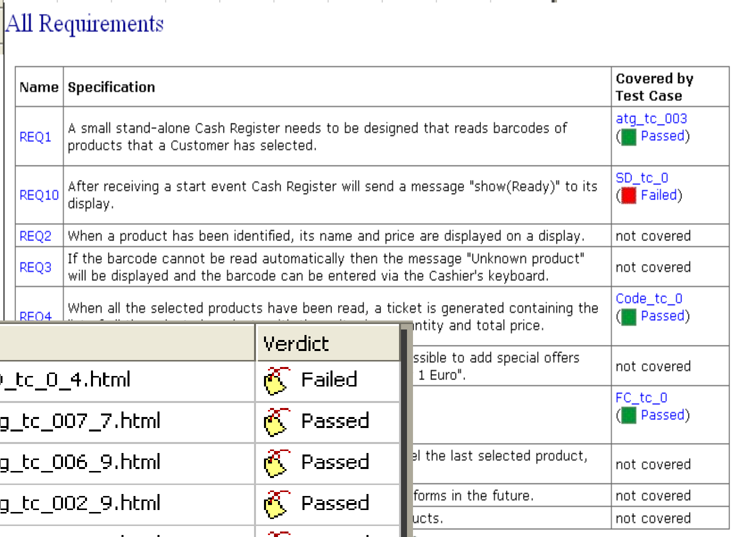
- Automated reporting of test results
  - Requirement to test coverage table
  - Test Coverage results
  - Complete test results in Rational Publishing Engine reports



To: Requirement Scope: CashRegister

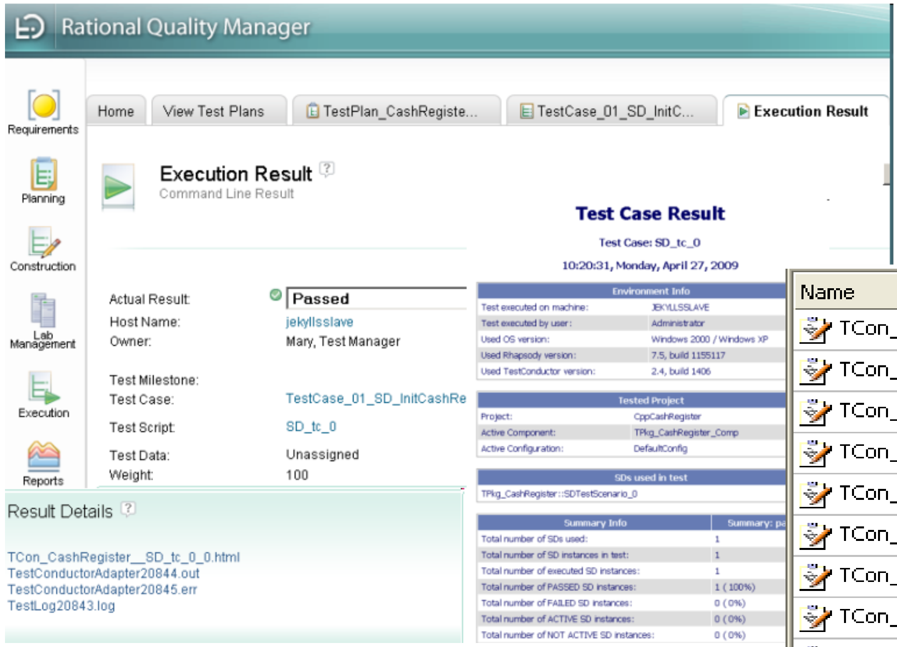
From: TestCase Scope: CashRegis

	REQ1	REQ2	REQ3	REQ4	REQ5	REQ6	REQ7	REQ8	REQ9	REQ10
SD_tc_0										REQ10
atg_tc_007										
atg_tc_006										
atg_tc_002										
atg_tc_003	REQ1									
atg_tc_004										
atg_tc_016										
atg_tc_017										



All Requirements

Name	Specification	Covered by Test Case
REQ1	A small stand-alone Cash Register needs to be designed that reads barcodes of products that a Customer has selected.	atg_tc_003 (🟩 Passed)
REQ10	After receiving a start event Cash Register will send a message "show(Ready)" to its display.	SD_tc_0 (🟥 Failed)
REQ2	When a product has been identified, its name and price are displayed on a display.	not covered
REQ3	If the barcode cannot be read automatically then the message "Unknown product" will be displayed and the barcode can be entered via the Cashier's keyboard.	not covered
REQ4	When all the selected products have been read, a ticket is generated containing the ntity and total price.	Code_tc_0 (🟩 Passed)
	ssible to add special offers 1 Euro".	not covered
		FC_tc_0 (🟩 Passed)
	el the last selected product,	not covered
	forms in the future.	not covered
	ucts.	not covered



Rational Quality Manager

Execution Result
Command Line Result

Actual Result:	Passed
Host Name:	jekyllsslave
Owner:	Mary, Test Manager
Test Milestone:	
Test Case:	TestCase_01_SD_InitCashRe
Test Script:	SD_tc_0
Test Data:	Unassigned
Weight:	100

Result Details

TCon_CashRegister__SD_tc_0_0.html
TestConductorAdapter20844.out
TestConductorAdapter20845.err
TestLog20843.log

Test Case Result

Test Case: SD_tc_0

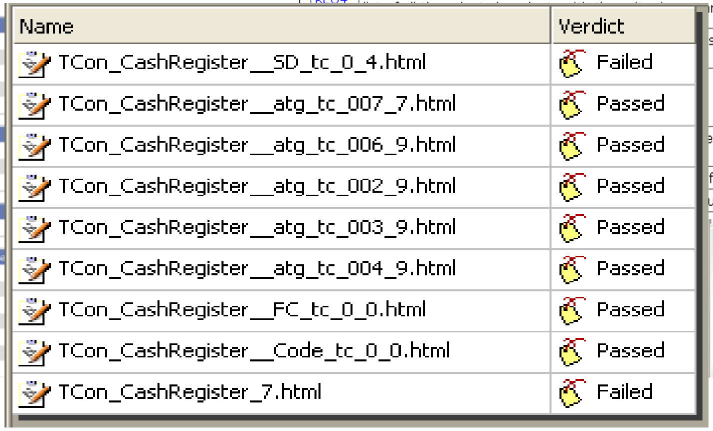
10:20:31, Monday, April 27, 2009

Environment Info	
Test executed on machine:	JEKYLLSSLAVE
Test executed by user:	Administrator
Used OS version:	Windows 2000 / Windows XP
Used Rhapsody version:	7.5, build 1155117
Used TestConductor version:	2.4, build 1406

Tested Project	
Project:	CppCashRegister
Active Component:	TPkg_CashRegister_Comp
Active Configuration:	DefaultConfig

SDs used in test
TPkg_CashRegister::SDTestScenario_0

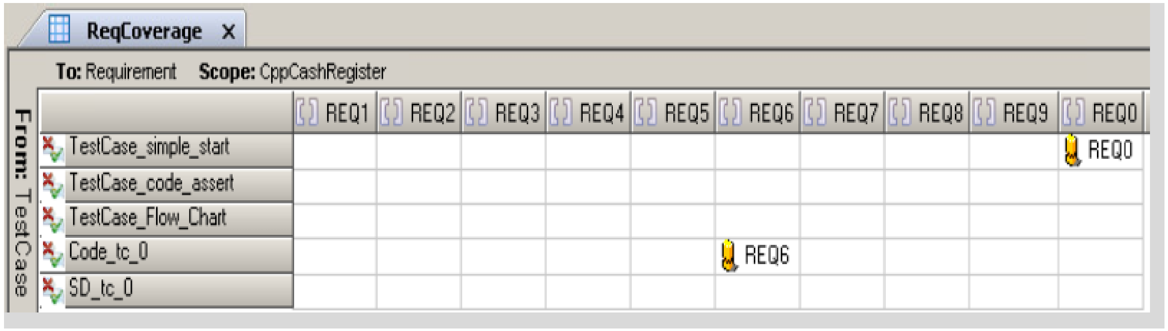
Summary Info	Summary: pa
Total number of SDs used:	1
Total number of SD instances in test:	1
Total number of executed SD instances:	1
Total number of PASSED SD instances:	1 (100%)
Total number of FAILED SD instances:	0 (0%)
Total number of ACTIVE SD instances:	0 (0%)
Total number of NOT ACTIVE SD instances:	0 (0%)



Name	Verdict
TCon_CashRegister__SD_tc_0_4.html	Failed
TCon_CashRegister__atg_tc_007_7.html	Passed
TCon_CashRegister__atg_tc_006_9.html	Passed
TCon_CashRegister__atg_tc_002_9.html	Passed
TCon_CashRegister__atg_tc_003_9.html	Passed
TCon_CashRegister__atg_tc_004_9.html	Passed
TCon_CashRegister__FC_tc_0_0.html	Passed
TCon_CashRegister__Code_tc_0_0.html	Passed
TCon_CashRegister_7.html	Failed

Coverage Analysis is one of the key benefits of automation

Which requirements are covered?



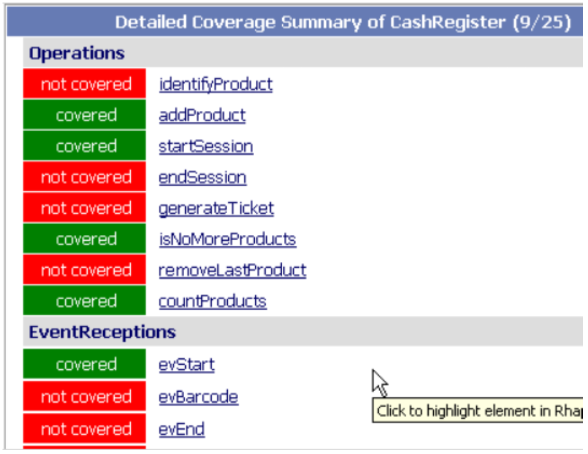
ReqCoverage

To: Requirement   Scope: CppCashRegister

From: TestCase

	REQ1	REQ2	REQ3	REQ4	REQ5	REQ6	REQ7	REQ8	REQ9	REQ0
TestCase_simple_start										REQ0
TestCase_code_assert										
TestCase_Flow_Chart										
Code_tc_0						REQ6				
SD_tc_0										

Which model elements are covered?

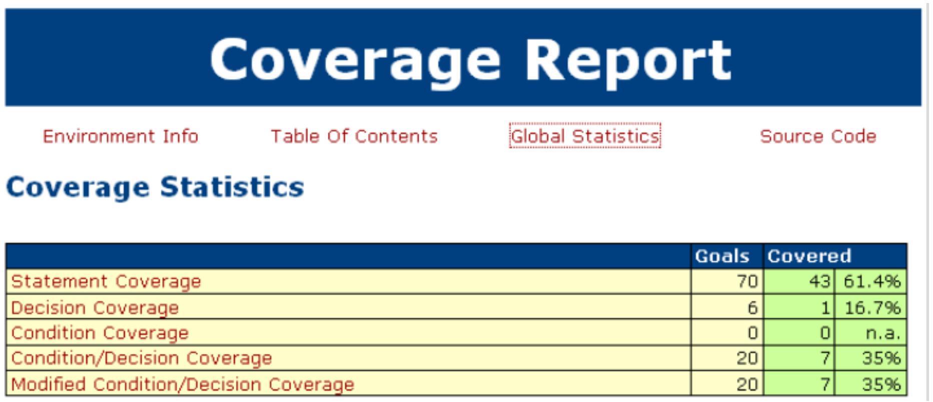


Detailed Coverage Summary of CashRegister (9/25)

Operations	
not covered	identifyProduct
covered	addProduct
covered	startSession
not covered	endSession
not covered	generateTicket
covered	isNoMoreProducts
not covered	removeLastProduct
covered	countProducts
EventReceptions	
covered	evStart
not covered	evBarcode
not covered	evEnd

Coverage Analysis is one of the key benefits of automation

What code is covered?

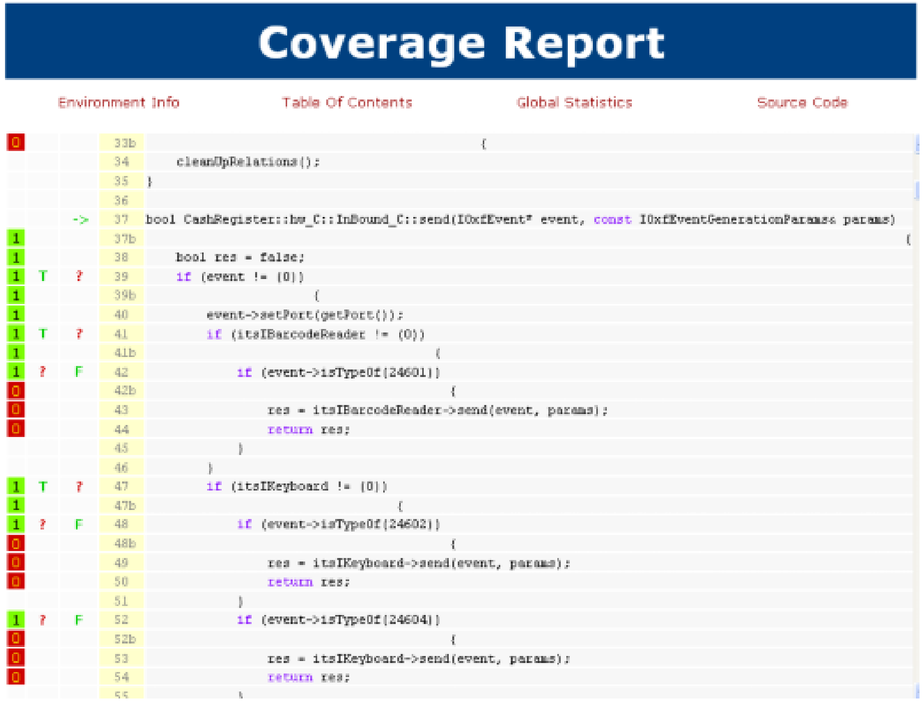


Coverage Report

Environment Info | Table Of Contents | Global Statistics | Source Code

Coverage Statistics

	Goals	Covered	
Statement Coverage	70	43	61.4%
Decision Coverage	6	1	16.7%
Condition Coverage	0	0	n.a.
Condition/Decision Coverage	20	7	35%
Modified Condition/Decision Coverage	20	7	35%



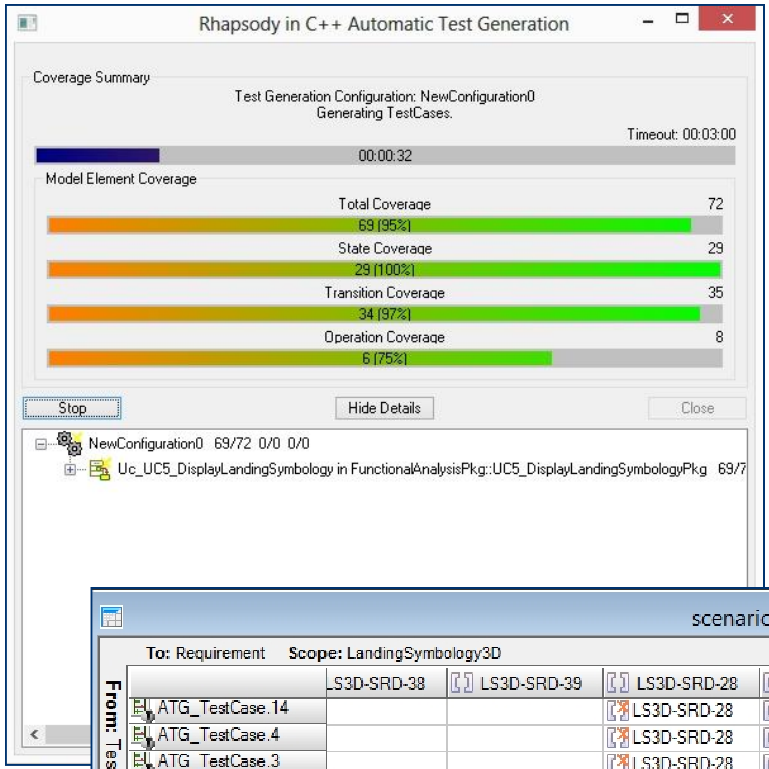
Coverage Report

Environment Info | Table Of Contents | Global Statistics | Source Code

```
0        33b                                          {
         34      cleanUpRelations();
         35  }
         36
      -> 37  bool CashRegister::hw_C::InBound_C::send(IOxfEvent* event, const IOxfEventGenerationParams& params)
1        37b                                                                                                  {
1        38      bool res = false;
1  T  ?  39      if (event != (0))
1        39b                       {
1        40          event->setPort(getPort());
1  T  ?  41          if (itsIBarcodeReader != (0))
1        41b                                   {
1  ?  F  42              if (event->isTypeOf(24601))
0        42b                                        {
0        43                  res = itsIBarcodeReader->send(event, params);
0        44                  return res;
         45              }
         46          }
1  T  ?  47          if (itsIKeyboard != (0))
1        47b                              {
1  ?  F  48              if (event->isTypeOf(24602))
0        48b                                        {
0        49                  res = itsIKeyboard->send(event, params);
0        50                  return res;
         51              }
1  ?  F  52              if (event->isTypeOf(24604))
0        52b                                        {
0        53                  res = itsIKeyboard->send(event, params);
0        54                  return res;
         55              }
```

MBT – Automatic Test Generation (ATG)

- Requirements-based test cases are generated with specified model and requirement coverage.



Rhapsody in C++ Automatic Test Generation

Coverage Summary

Test Generation Configuration: NewConfiguration0
Generating TestCases.

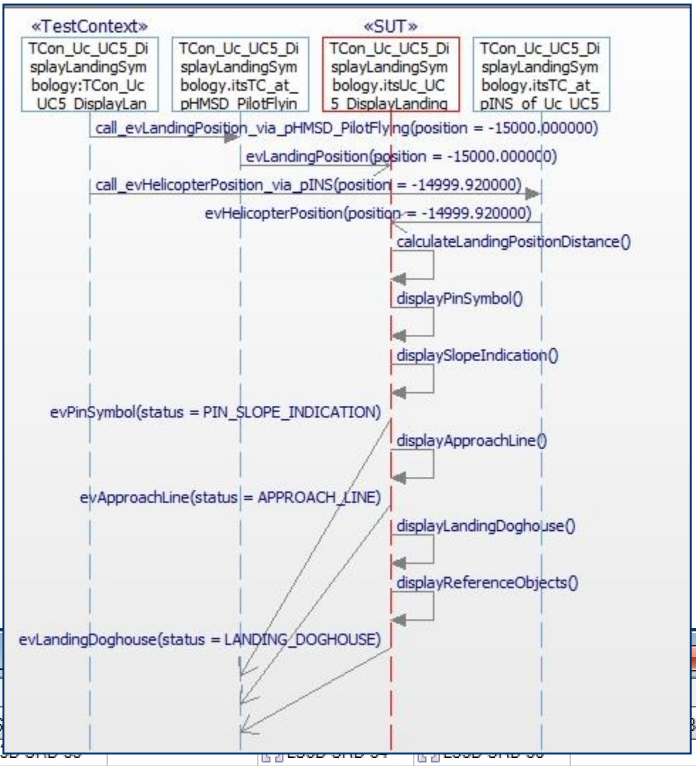
Timeout: 00:03:00

00:00:32

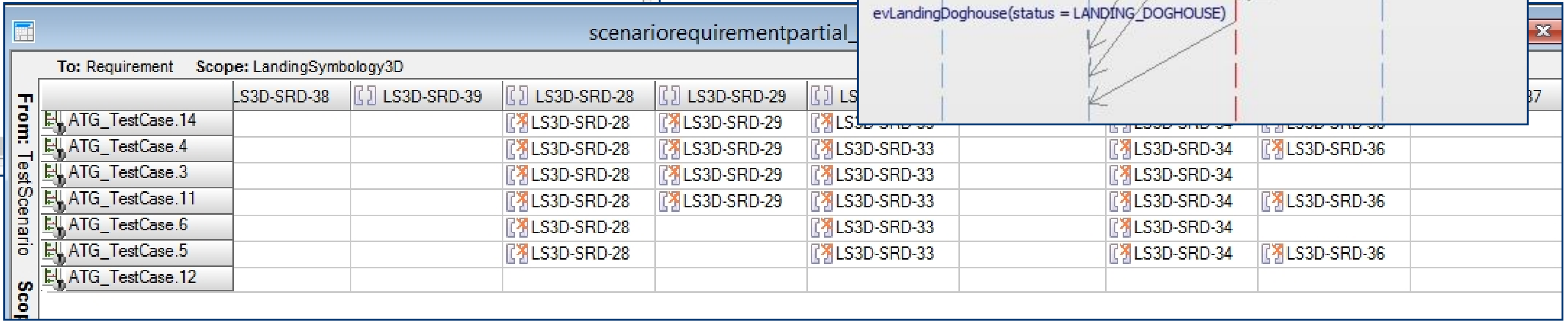
Model Element Coverage

Total Coverage	69 (95%)	72
State Coverage	29 (100%)	29
Transition Coverage	34 (97%)	35
Operation Coverage	6 (75%)	8

NewConfiguration0 69/72 0/0 0/0
Uc_UC5_DisplayLandingSymbology in FunctionalAnalysisPkg::UC5_DisplayLandingSymbologyPkg 69/7



«TestContext»
«SUT»
TCon_Uc_UC5_DisplayLandingSymbology:TCon_Uc_UC5_DisplayLan
TCon_Uc_UC5_DisplayLandingSymbology.itsTC_at_pHMSD_PilotFlyin
TCon_Uc_UC5_DisplayLandingSymbology.itsUc_UC5_DisplayLanding
TCon_Uc_UC5_DisplayLandingSymbology.itsTC_at_pINS_of_Uc_UC5
call_evLandingPosition_via_pHMSD_PilotFlying(position = -15000.000000)
evLandingPosition(position = -15000.000000)
call_evHelicopterPosition_via_pINS(position = -14999.920000)
evHelicopterPosition(position = -14999.920000)
calculateLandingPositionDistance()
displayPinSymbol()
displaySlopeIndication()
evPinSymbol(status = PIN_SLOPE_INDICATION)
displayApproachLine()
evApproachLine(status = APPROACH_LINE)
displayLandingDoghouse()
displayReferenceObjects()
evLandingDoghouse(status = LANDING_DOGHOUSE)



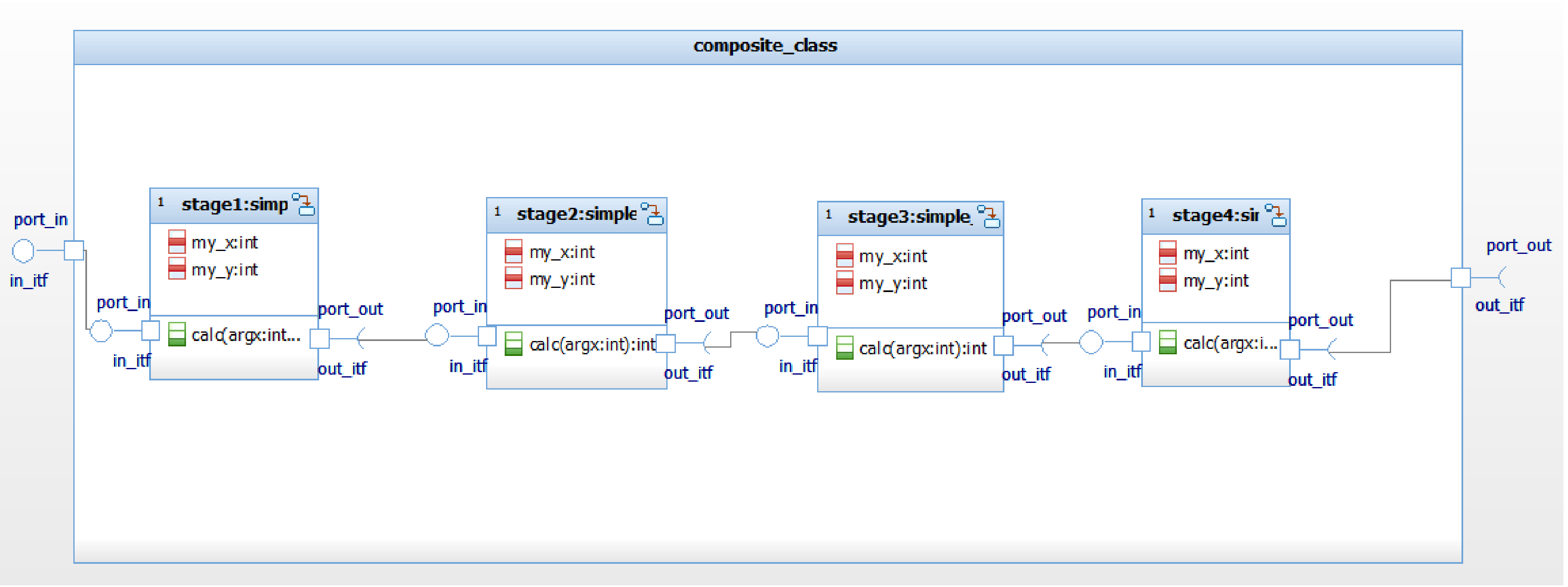
scenariorequirementpartial_

To: Requirement **Scope:** LandingSymbology3D

From: TestScenario **Scop**

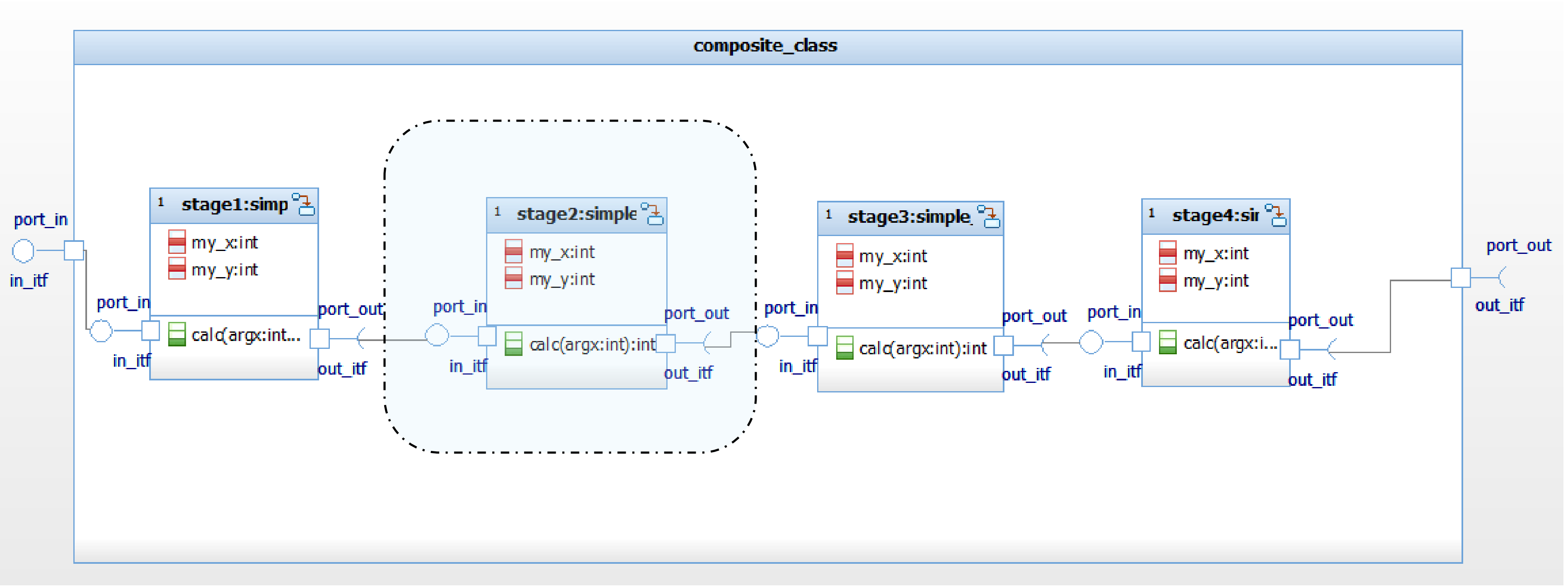
	LS3D-SRD-38	LS3D-SRD-39	LS3D-SRD-28	LS3D-SRD-29	LS				37
ATG_TestCase.14			LS3D-SRD-28	LS3D-SRD-29	LS3D-SRD-33		LS3D-SRD-34	LS3D-SRD-36	
ATG_TestCase.4			LS3D-SRD-28	LS3D-SRD-29	LS3D-SRD-33		LS3D-SRD-34	LS3D-SRD-36	
ATG_TestCase.3			LS3D-SRD-28	LS3D-SRD-29	LS3D-SRD-33		LS3D-SRD-34		
ATG_TestCase.11			LS3D-SRD-28	LS3D-SRD-29	LS3D-SRD-33		LS3D-SRD-34	LS3D-SRD-36	
ATG_TestCase.6			LS3D-SRD-28		LS3D-SRD-33		LS3D-SRD-34		
ATG_TestCase.5			LS3D-SRD-28		LS3D-SRD-33		LS3D-SRD-34	LS3D-SRD-36	
ATG_TestCase.12									

Sample System to demo MBT



- System shows an explicitly modeled input and output interface using ports
- System contains four units with explicitly modeled input and output interfaces using ports; the units get input integer values and multiply with 2
- Software architecture shows how the units are integrated using ports and links

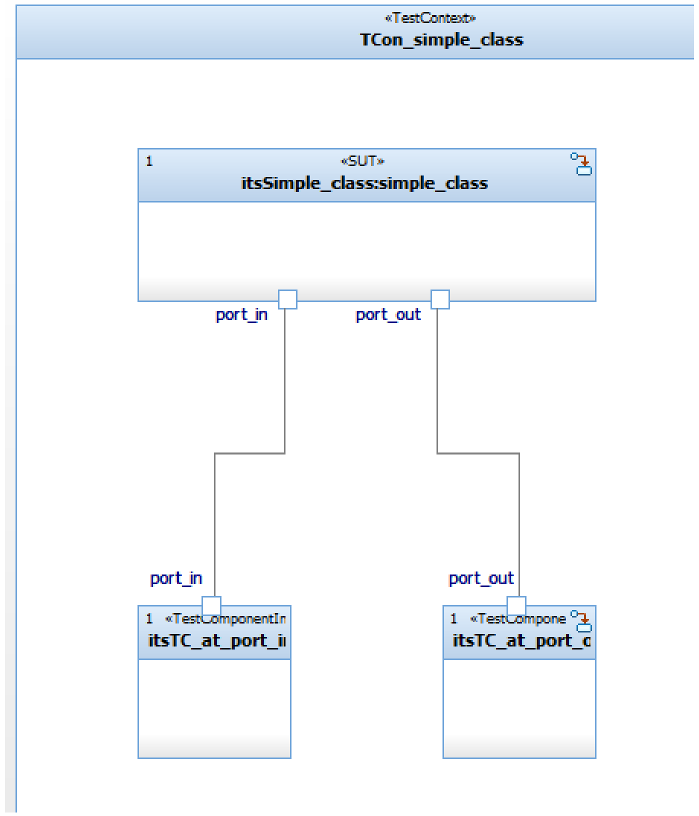
MBT: Unit Testing I

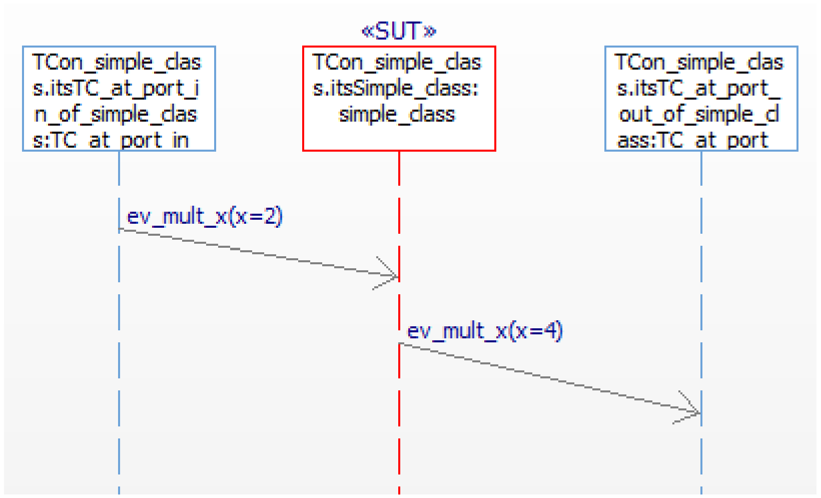


- Objective is to test each unit in isolation
- TestCondcutor automatically creates test architectures for each unit (SUT)
- “White box test”:
  - requirements based testing using the interfaces of the SUT
  - code coverage measurement of the internal structure of the SUT

InternetofThings

MBT: Unit Testing II



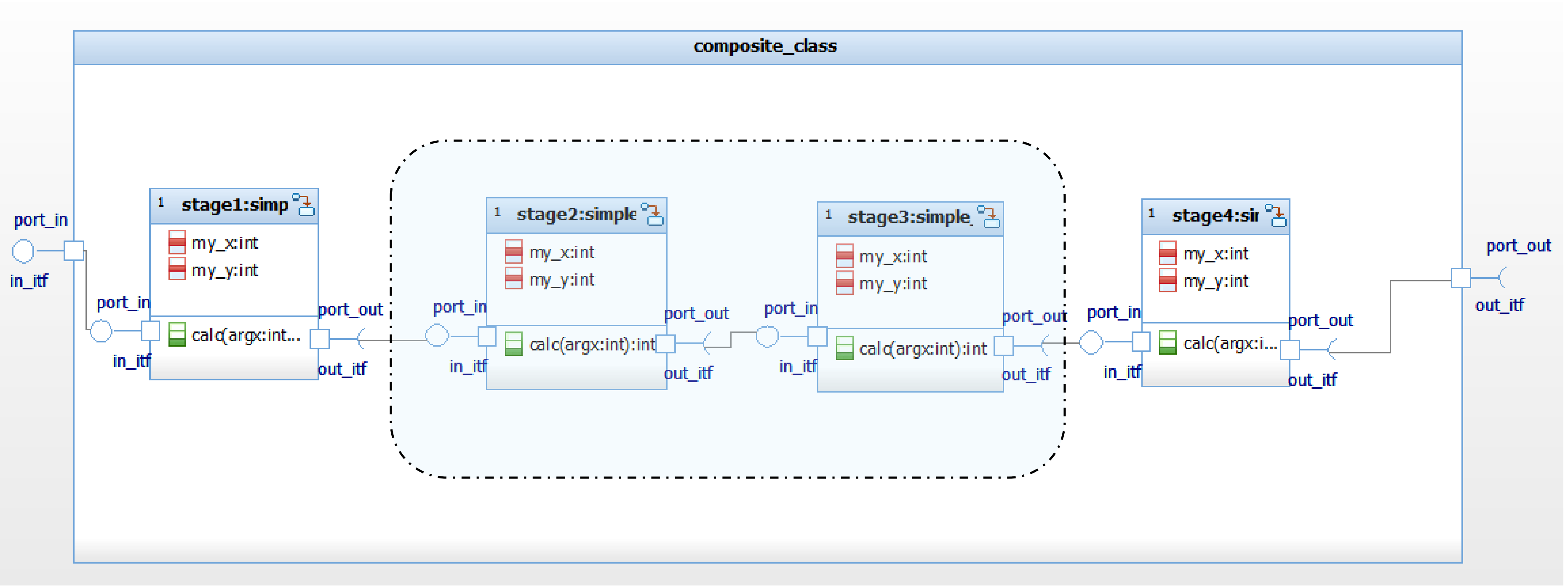

«SUT»
TCon_simple_class.itsTC_at_port_in_of_simple_class:TC_at_port_in
TCon_simple_class.itsSimple_class:simple_class
TCon_simple_class.itsTC_at_port_out_of_simple_class:TC_at_port
ev_mult_x(x=2)
ev_mult_x(x=4)

– An instance of the unit under test (SUT) is contained in the test architecture, and two test components which are connected to the ports of the SUT

– Developers specify the expected input / output behaviour in a test case

– TestConductor executes the unit tests and computes test verdicts (pass/fail)

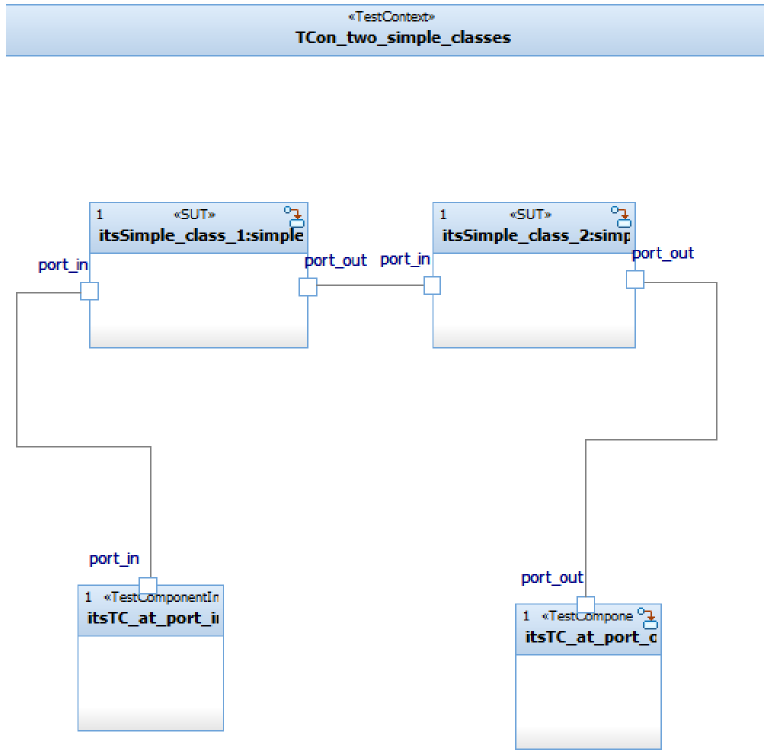
MBT: IntegrationTesting I

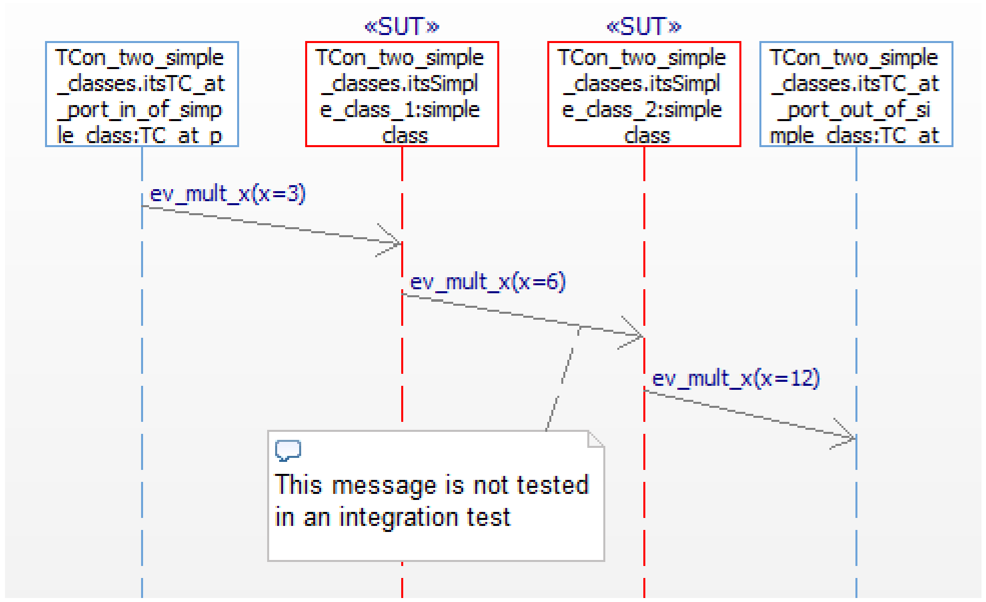


- Objective is to test two or more integrated units
- TestCondcutor automatically creates test architectures for one unit, developers can extend the test architecture to add more units (SUT)
- “Grey box test”
  - requirements based testing using the *external* interfaces of the integrated SUT
  - code coverage measurement of the internal structure of the SUT

InternetofThings

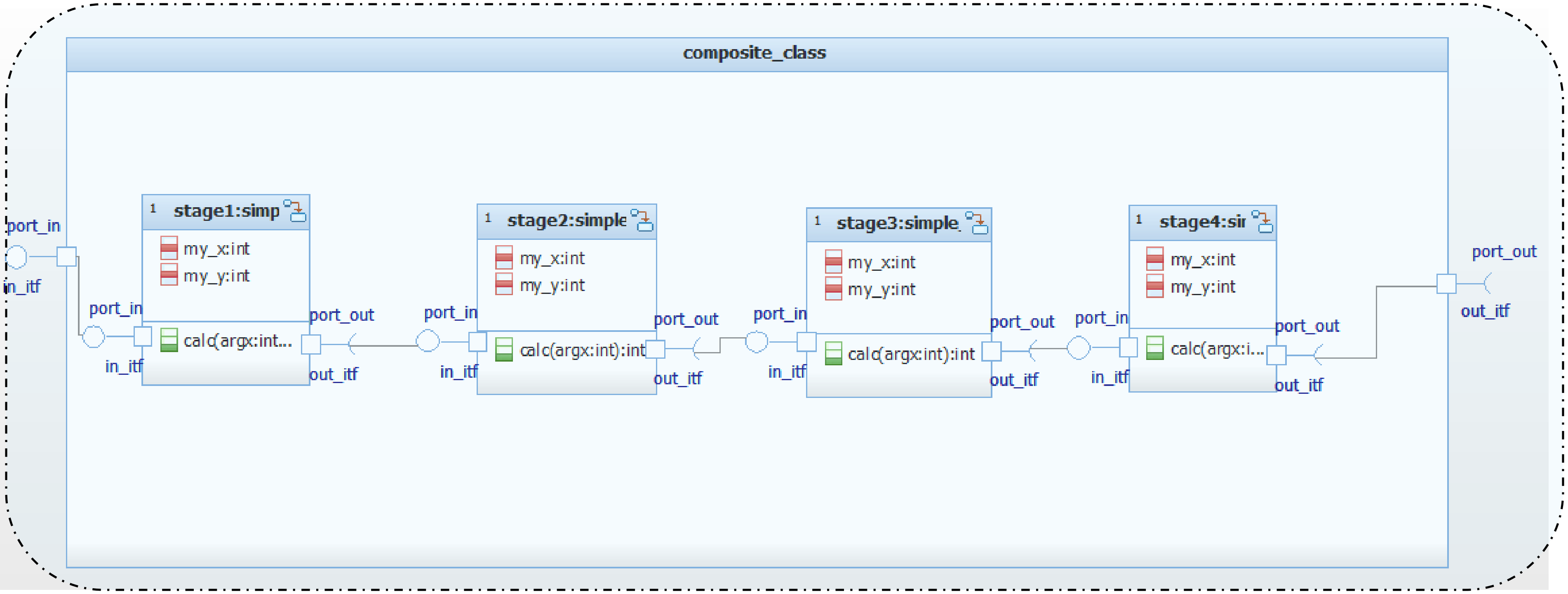
MBT: Integration Testing II


«TestContext»
TCon_two_simple_classes
1 «SUT» itsSimple_class_1:simple
1 «SUT» itsSimple_class_2:simp
port_in
port_out
port_in
port_out
port_in
port_out
1 «TestComponentIn itsTC_at_port_i
1 «TestCompone itsTC_at_port_c


TCon_two_simple_classes.itsTC_at_port_in_of_simple_class:TC_at_p
«SUT» TCon_two_simple_classes.itsSimple_class_1:simple class
«SUT» TCon_two_simple_classes.itsSimple_class_2:simple class
TCon_two_simple_classes.itsTC_at_port_out_of_simple_class:TC_at
ev_mult_x(x=3)
ev_mult_x(x=6)
ev_mult_x(x=12)
This message is not tested in an integration test

- Instances of the two units under test (SUT) are contained in the test architecture, and two test components which are connected to the ports of the SUT
- Developers specify the expected input / output behaviour of the integrated units
- TestConductor executes the integration tests and computes test verdicts (pass/fail)

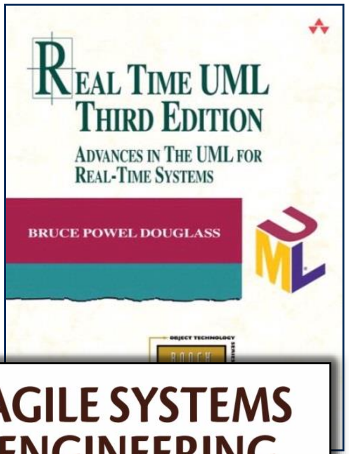
MBT: Software System Testing I

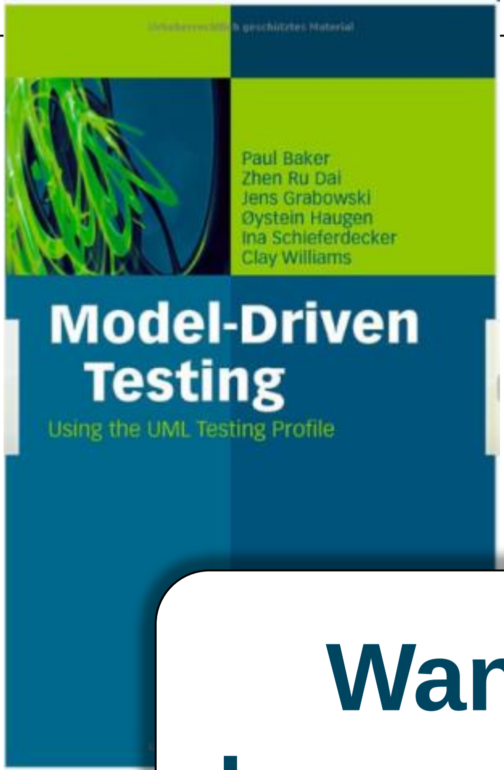


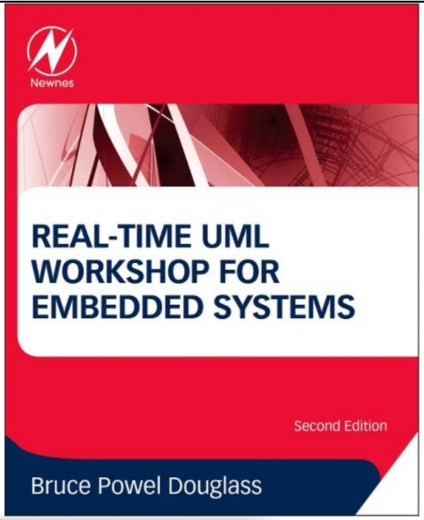
- Objective is to test the whole SW system on host *or on an embedded target*
- TestCondcutor automatically creates test architectures for the SW system using the system ports and interfaces
- “Black box test”
    - requirements based testing using the interfaces of the SUT

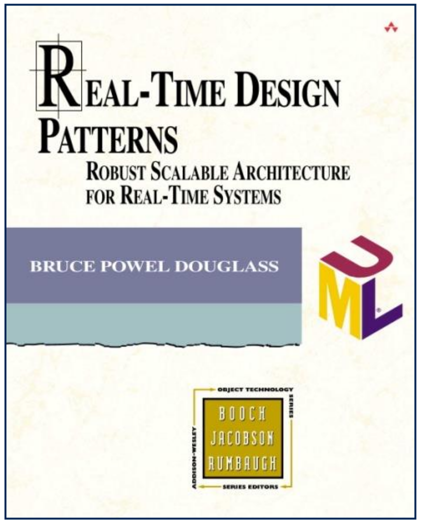
Summary

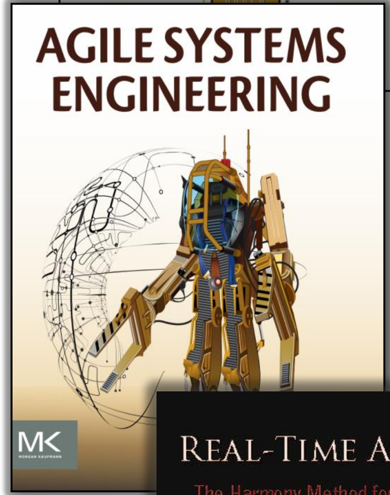
- Testing is hard!
- Models are simplifications of reality that allow us to focus on relevant issues
- Models provide significant enhancement to our ability to deal with engineering data, such as requirements, design, and implementation
- Models likewise enhance our ability to test:
  - Development of test architectures from model structures
  - Development and representation of test cases
  - Execution of test cases against the SUT in the test architecture
  - Computation of verdicts (pass/fail)
  - Determination of coverage (model and/or code)
- The UML Testing Profile defines a standard way for modeling test-related information
- Model-Based Testing can be done
  - Manually by “instrumenting” actors or creation of testing stubs
  - Automatically with tools such as Test Conductor
- Automation of Model Based Testing provides real benefits
  - Repeatable testing
  - Auto generation of test architectures
  - Auto execution of test suites and analysis of outcomes to determine verdicts
  - ATG can even analyze model structures and create test cases to ensure coverage

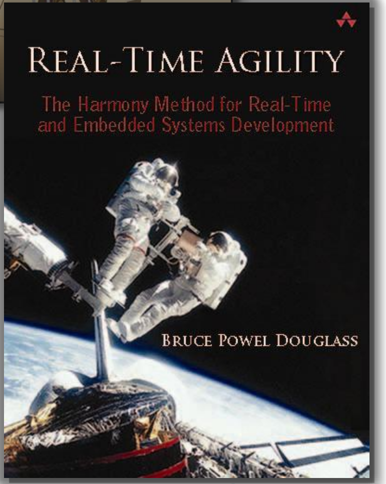

REAL TIME UML
THIRD EDITION
ADVANCES IN THE UML FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS


Paul Baker
Zhen Ru Dai
Jens Grabowski
Øystein Haugen
Ina Schieferdecker
Clay Williams
Model-Driven Testing
Using the UML Testing Profile

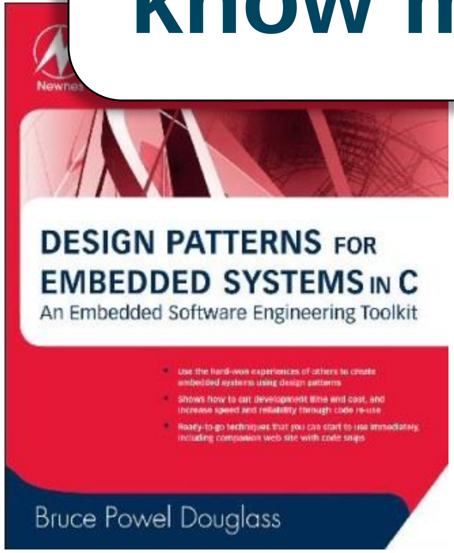

Newnes
REAL-TIME UML WORKSHOP FOR EMBEDDED SYSTEMS
Second Edition
Bruce Powel Douglass


REAL-TIME DESIGN PATTERNS
ROBUST SCALABLE ARCHITECTURE FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS
BOOCH JACOBSON RUMBAUGH

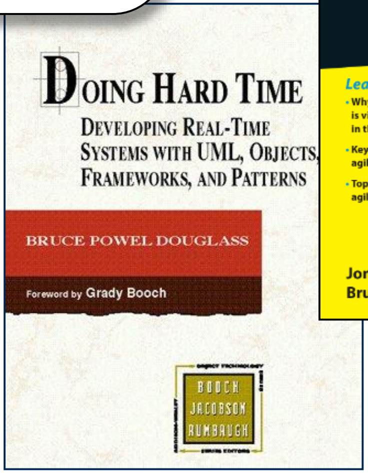

AGILE SYSTEMS ENGINEERING

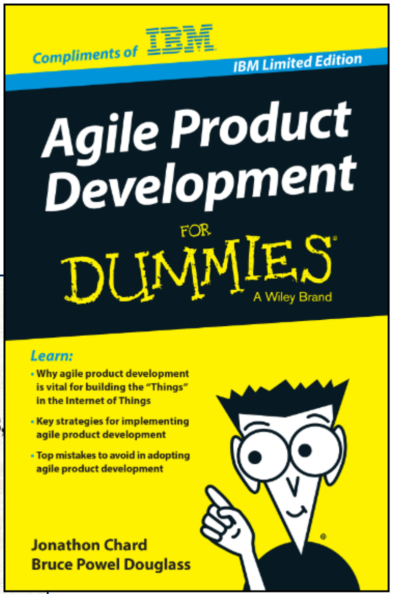

REAL-TIME AGILITY
The Harmony Method for Real-Time and Embedded Systems Development
BRUCE POWEL DOUGLASS

Want to know more?


Newnes
DESIGN PATTERNS FOR EMBEDDED SYSTEMS IN C
An Embedded Software Engineering Toolkit
- Use the hard-won experiences of others to create embedded systems using design patterns
- Shows how to cut development time and cost, and increase speed and reliability through code re-use
- Ready-to-go techniques that you can start to use immediately, including companion web site with code snips

Bruce Powel Douglass


DOING HARD TIME
DEVELOPING REAL-TIME SYSTEMS WITH UML, OBJECTS, FRAMEWORKS, AND PATTERNS
BRUCE POWEL DOUGLASS
Foreword by Grady Booch
BOOCH JACOBSON RUMBAUGH


Compliments of IBM
IBM Limited Edition
Agile Product Development for DUMMIES
A Wiley Brand
Learn:
- Why agile product development is vital for building the "Things" in the Internet of Things
- Key strategies for implementing agile product development
- Top mistakes to avoid in adopting agile product development

Jonathon Chard
Bruce Powel Douglass