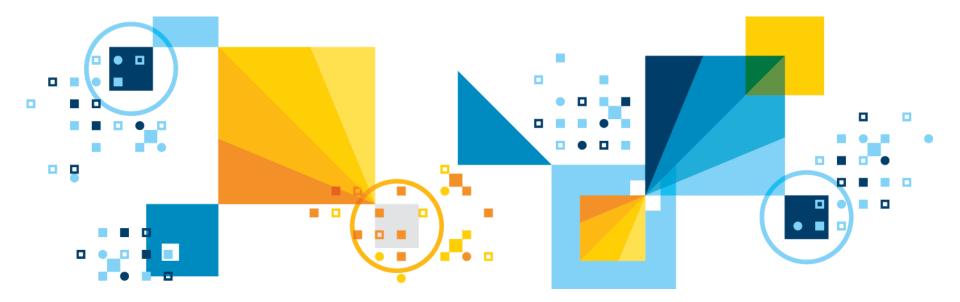
## What is Model-Based Testing ... and how do I get started?

#### Bruce Powel Douglass, Ph.D.

<u>www.bruce-douglass.com</u> Bruce.Douglass@outlook.com

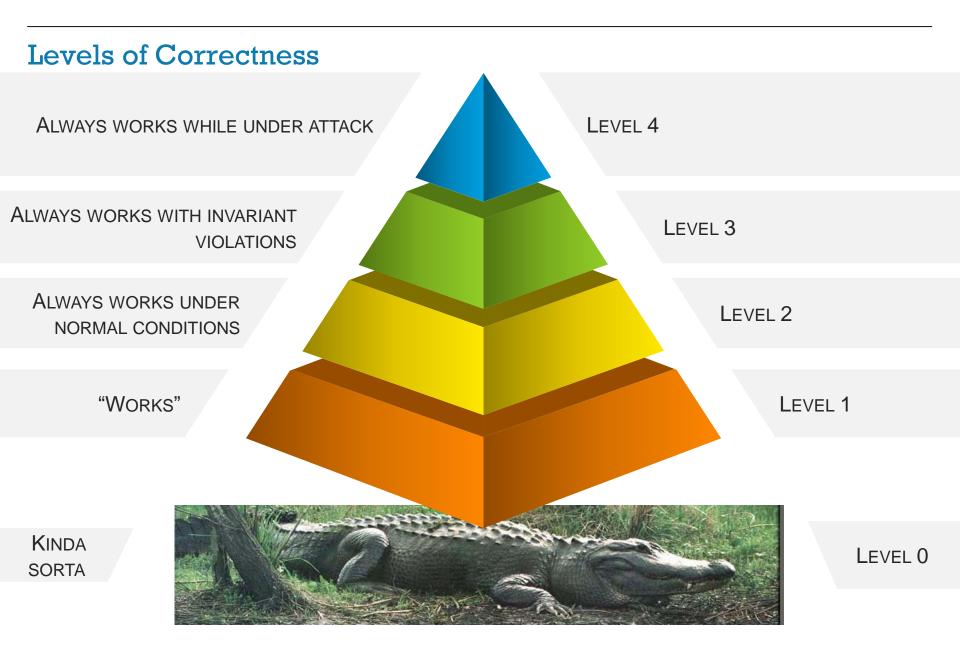


© 2020 Bruce Douglass

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.



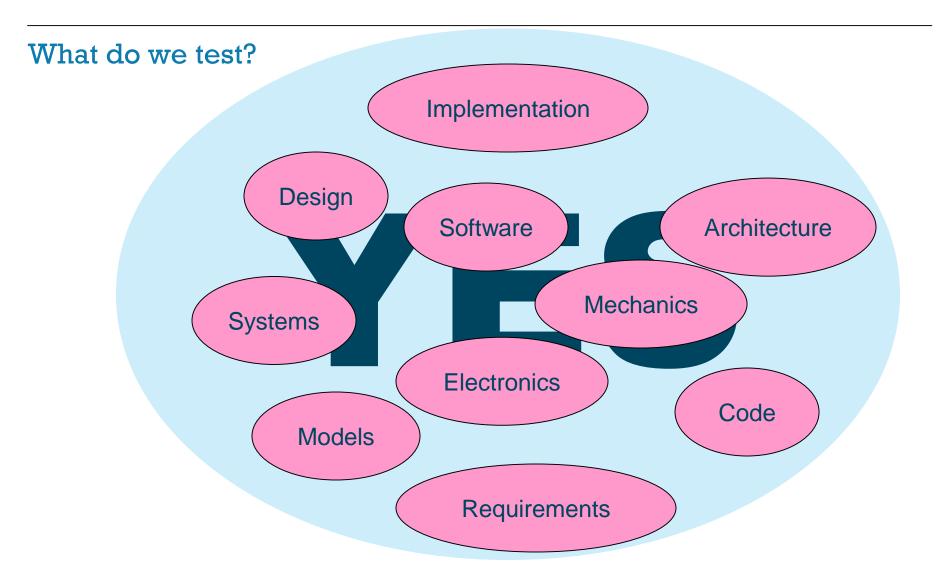
#### Why do we test?

To uncover limitations and constraints

To find out if it works

To find out if it DOESN'T work

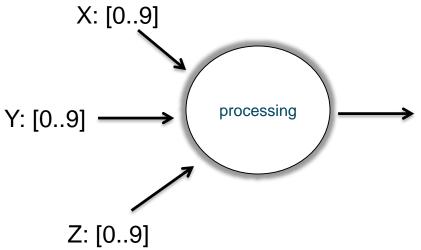
To demonstrate compliance



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



Testing can never be complete – there are an essentially infinite set of combinations of value, sequence, and timing At first look, this has 1000 combinations to be tested. But what if

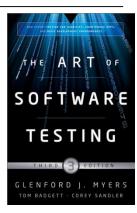
- X comes before Y? Or Z before X?
- The system expects Z to occur in < 20ms but it arrives at 30ms?
- The output comes too late?
- What if Z, Y, and Z are not independent?
   Example: if X>5 then Y must be <= 2</li>
- What if X is -1?
- Does the case Z==-20 fail in the same way as X == 45?
- What if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

#### **Glenford Meyer's** The Art of Testing

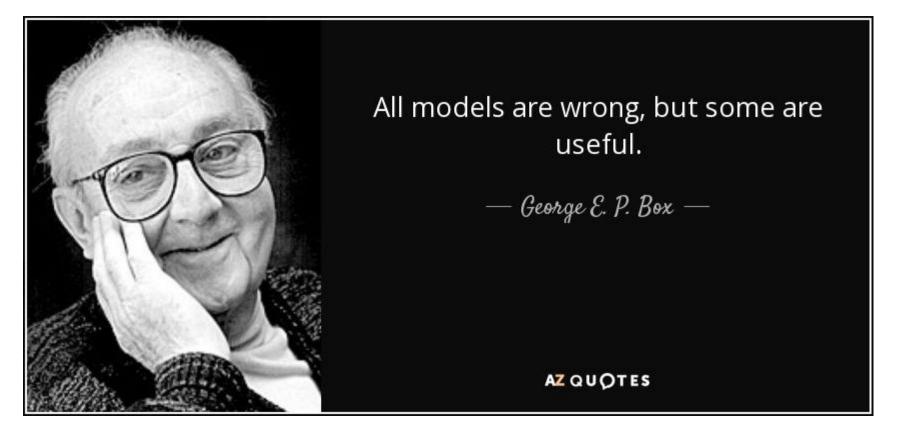
- Consider the simple problem
  - The program reads three integer values from a text input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.

#### 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

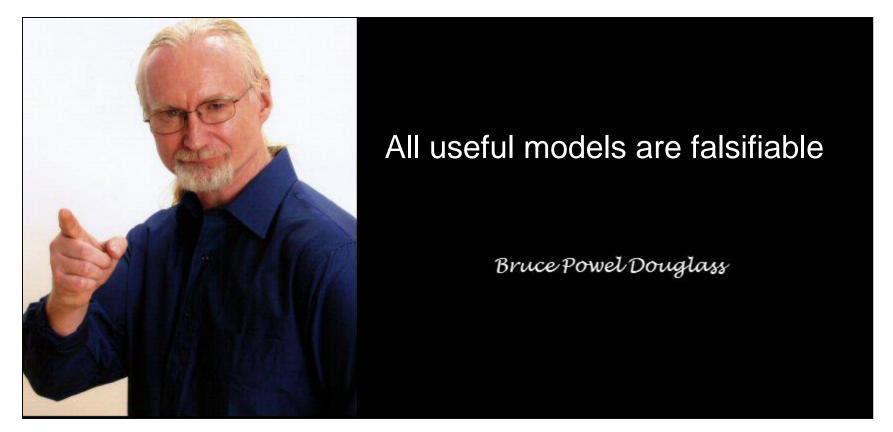


#### Models

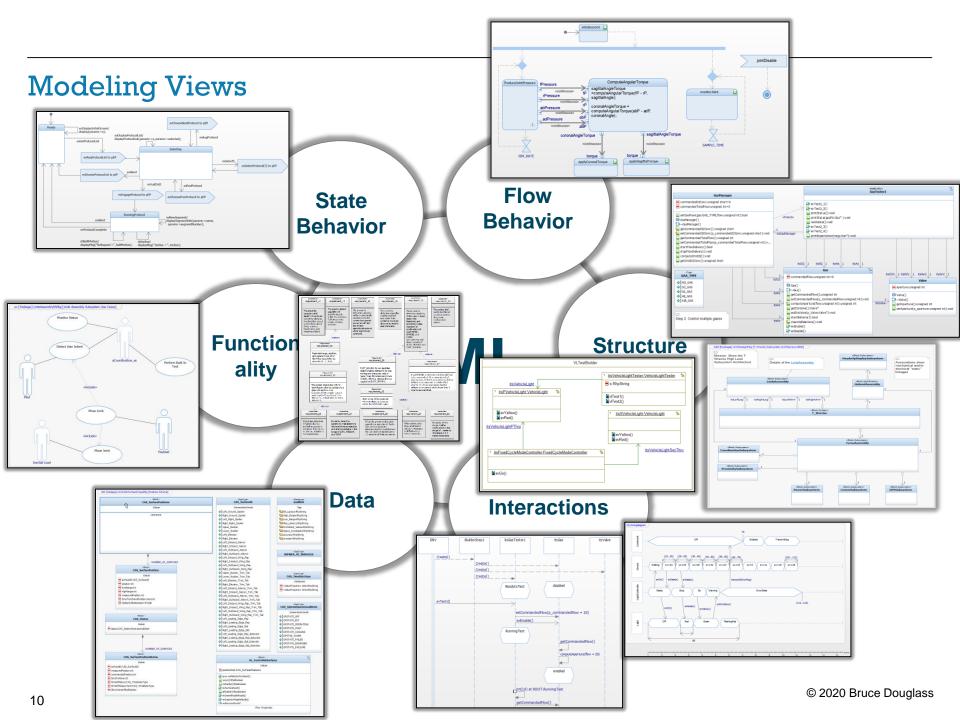


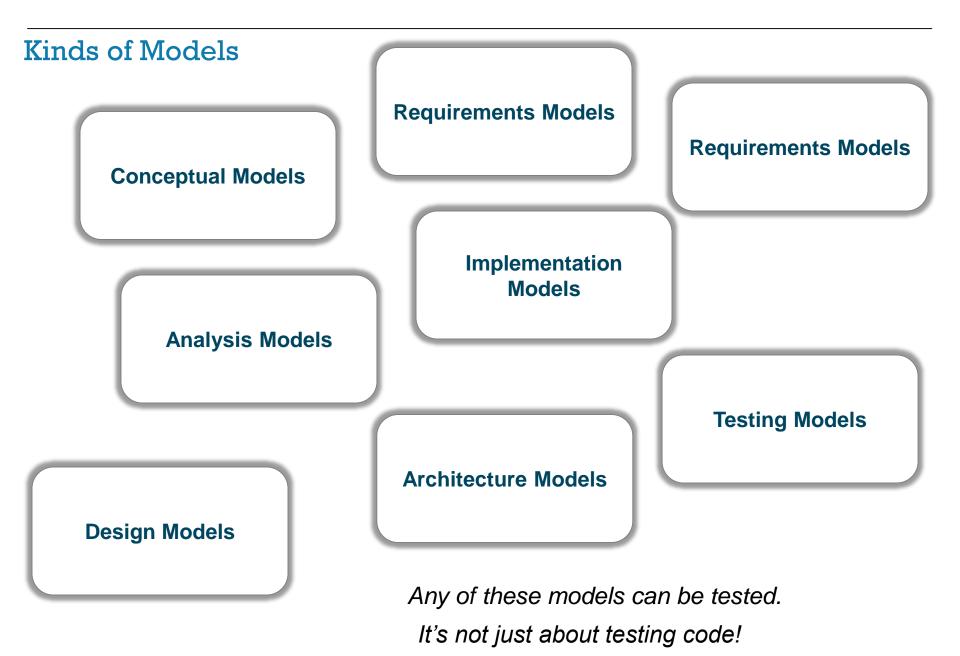
- Problem: Reality is too complex
- Solution: Create a model
- A model is always a simplification of reality, wherein we focus on aspects relevant to things we care about and elide details of those things we do not.

#### 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





#### 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?

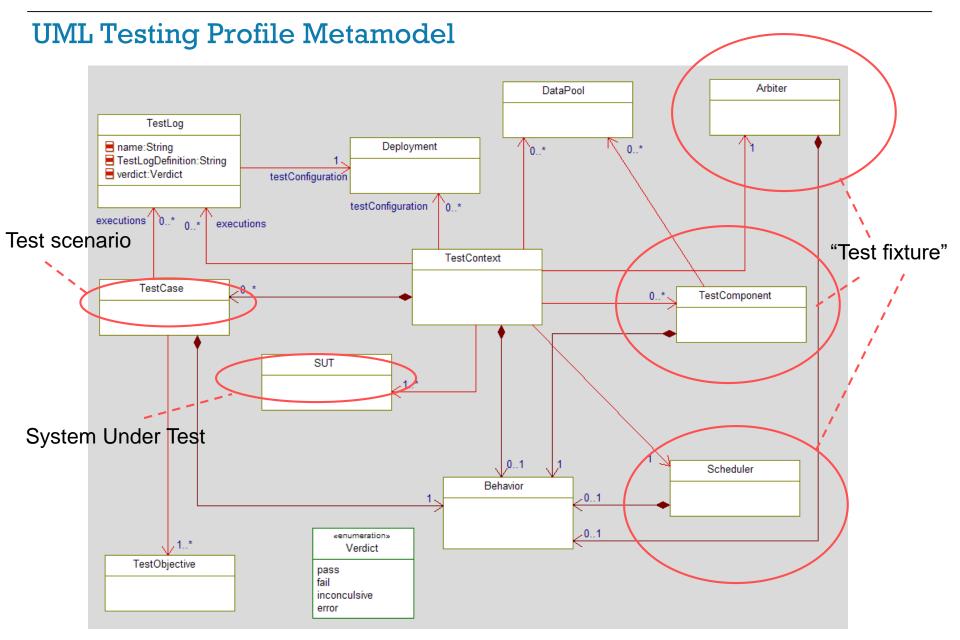
- Creation of Test Architecture
- Capturing of outcomes during execution
- Conversion of requirements scenarios to test cases
- Application of test cases to system
- Identification of points of failure
- Gathering of pass/fail statistics
- Computation of coverage metrics

#### **UML Testing Profile**

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

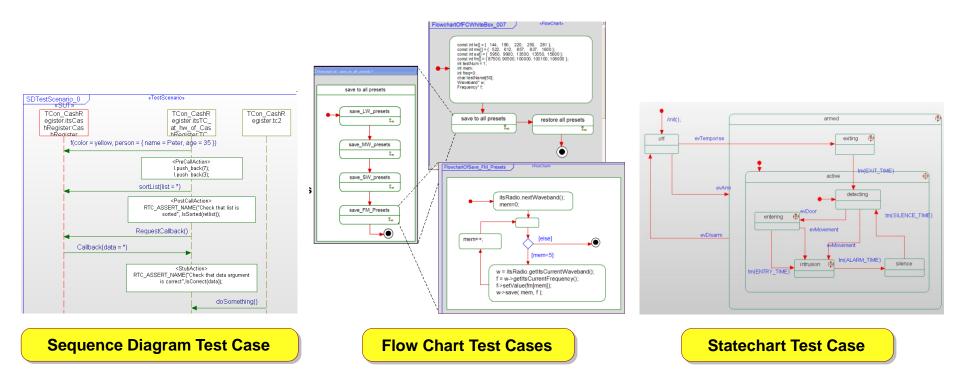
The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems. It is a test modeling language that can be used with all major object and component technologies and applied to testing systems in various application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

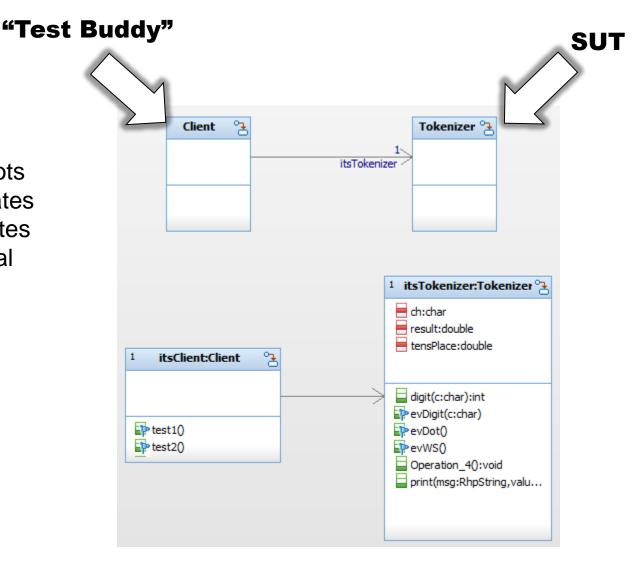


#### Capture test cases with UML/SysML

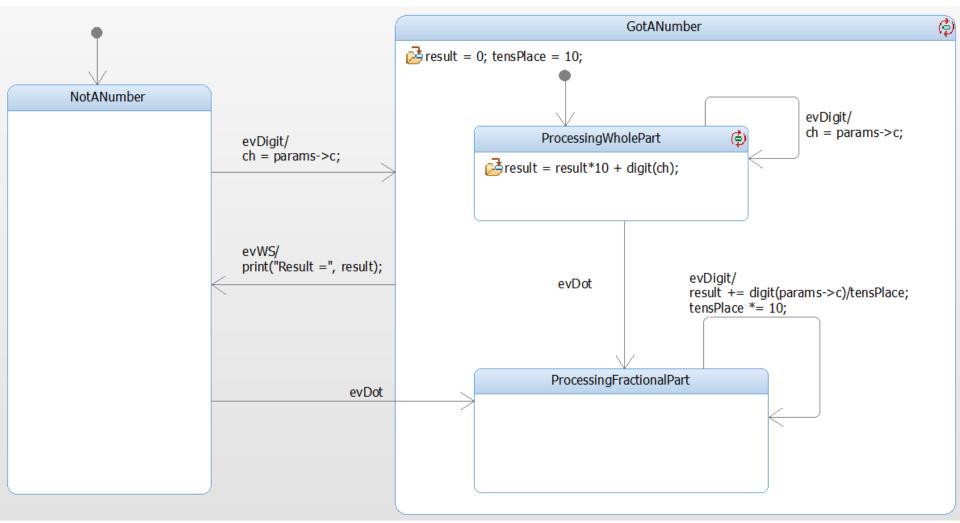
- Recommend using OMG's standard UML Testing Profile (<u>www.omg.org</u>)
- Specify test cases visually for better communication across teams
- Creating code tests cases or importing Cunit/Cpp unit tests also possible
- Can be done manually or with automation (via Test Conductor)



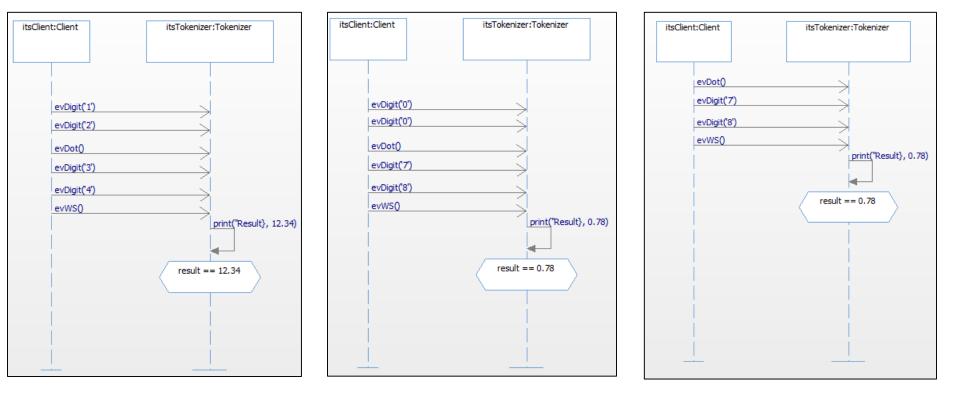
This simple model receives digits and dots as characters, evaluates the string and computes the corresponding real value



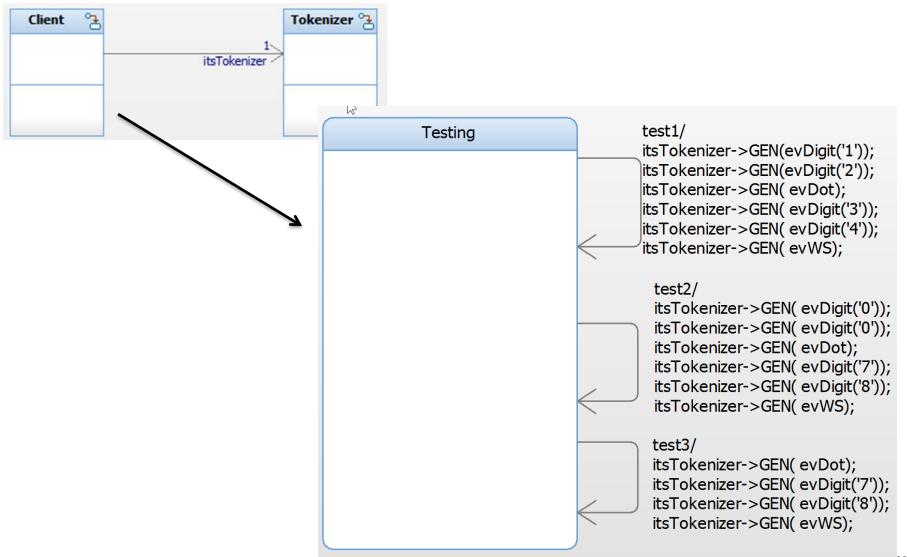
This is the state machine for the Tokenizer class



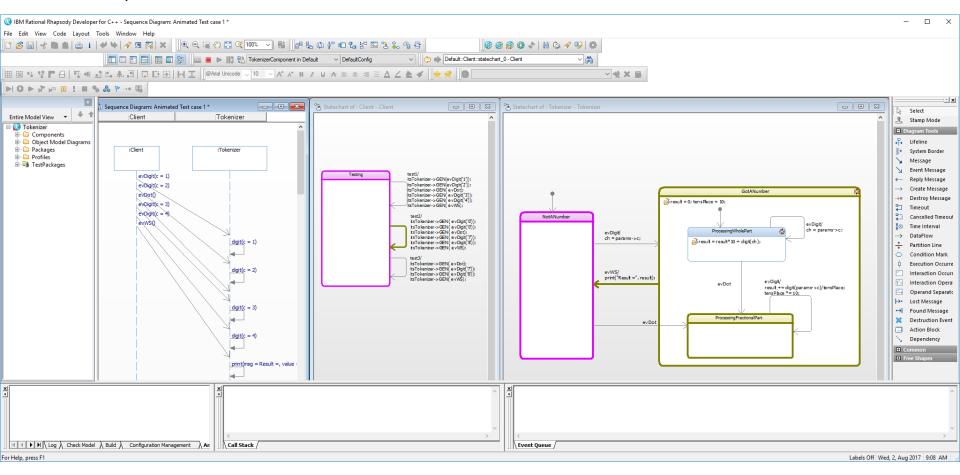
#### Create Test Cases as Sequence Diagrams



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

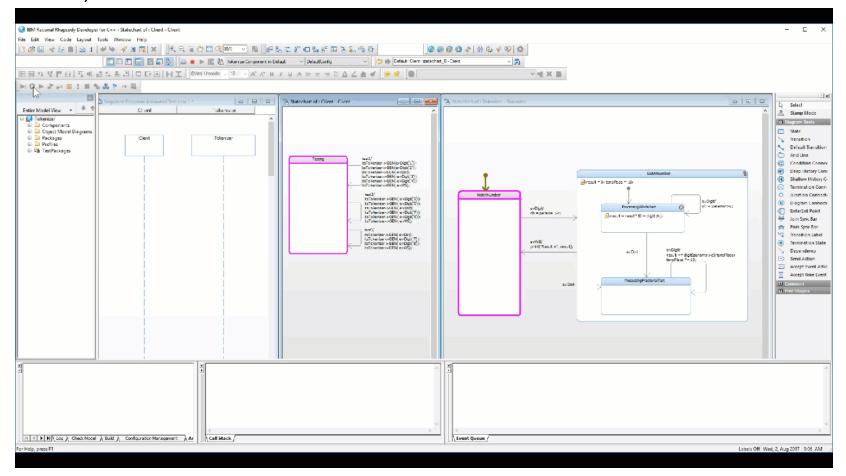


Now execute the model and create "animated sequence diagrams"\* from the execution)



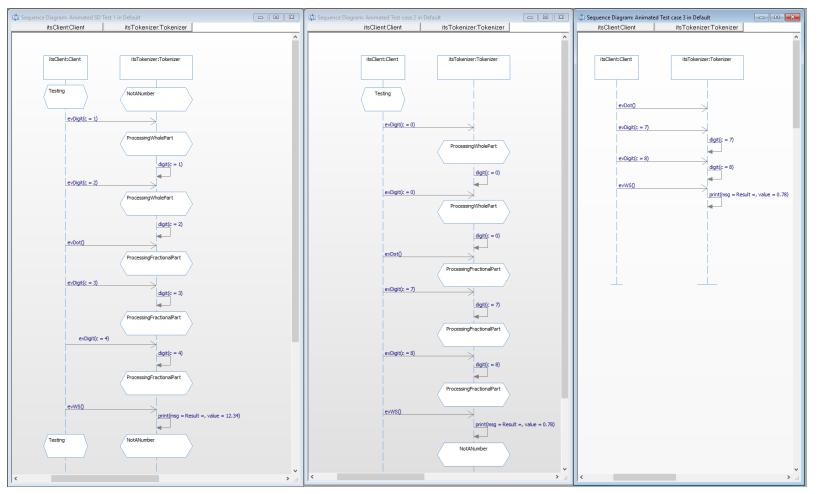
\* Rhapsody feature – can produce sequence diagrams from the interaction of modelled elements during execution

Now execute the model and create "animated sequence diagrams"\* from the execution)



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

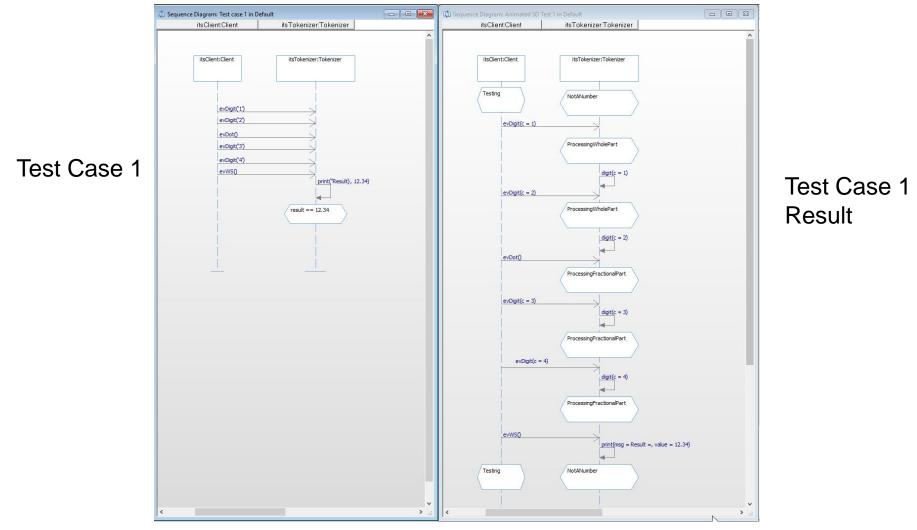
Review the outcomes and compare to the test specifications



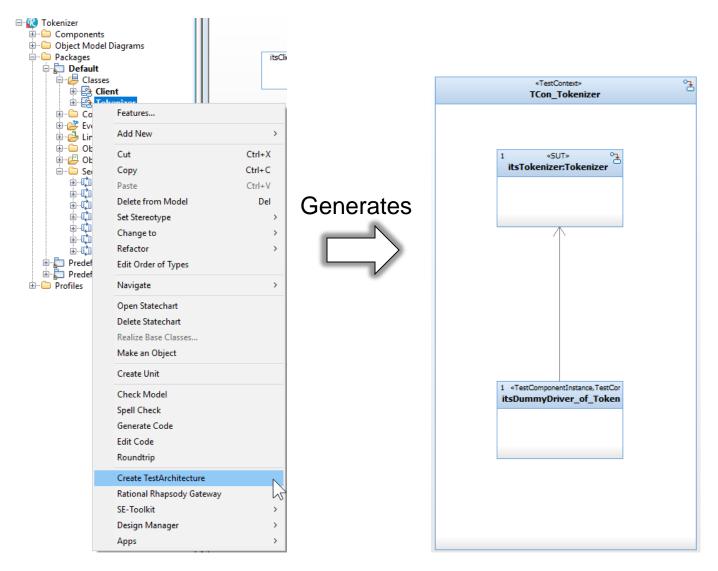
Test Case 1 Outcome

Test Case 2 Outcome Test Case 3 Outcome

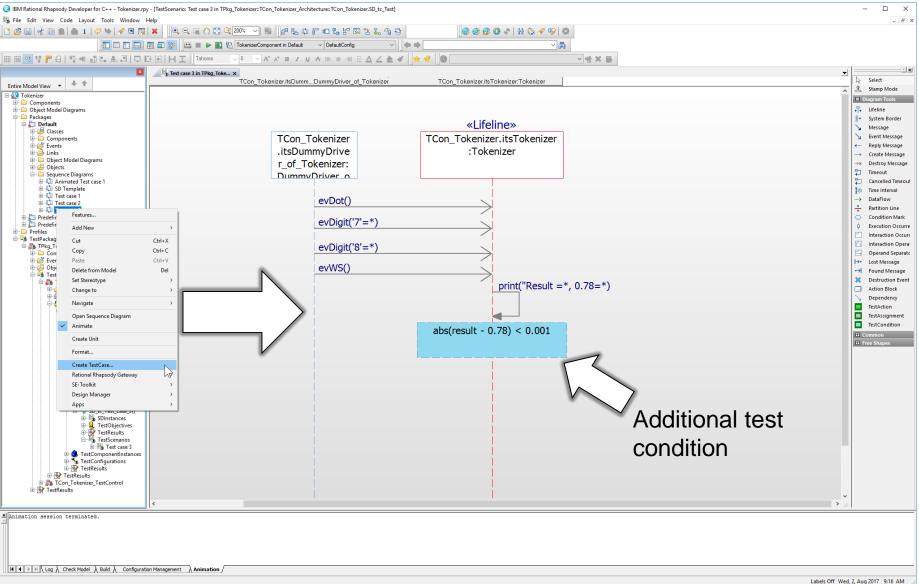
Review the outcomes and compare to the test specifications



#### Example Model: Tokenizer (Test Conductor)



#### Example Model: Tokenizer (Test Conductor)



#### Example Model: Tokenizer (Test Conductor)

🔲 🗈 📰 🖶 🐌 🛗 🐘 🖏 TPkg\_Tokenizer\_Comp in TPkg\_Tok 🗸 DefaultConfig 🗸 🔅 TPkg\_Tokenizer\_Architecture::TCon\_To 🗸 🍰

## Test outcomes

High File Edit View Code Layout Tools Window Help

Packages

😢 IBM Rational Rhapsody Developer for C++ - Tokenizer.rpy - [TestScenario: Test case 3 in TPkg\_Tokenizer: TCon\_Tokenizer\_Architecture:: TCon\_Tokenizer.SD\_tc\_Test]

3 & 데 생 @ ▲ ▲ ↓ / / / 제 정 🗶 🔍 이 때 신 전 이 2001 🗸 등 👘 등 다 (\* 대 등 방 전 등 등 다 )

#### **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

${\sf TPkg\_Tokenizer::} {\sf TCon\_Tokenizer\_Architecture::} {\sf TCon\_Tokenizer.SD\_tc\_0::} {\sf TC} {\sf Test}$
case 1

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

Detailed Assertion Informa	tion
== 12.34	PASSED

**Result Verification** 

Result verification successful

#### TestCase: SD\_tc\_Test\_case\_1

SequenceDiagram used in TestCase

TPka Takenizer TCan Takenizer Architecture TCan Takenizer SD to Test case 1"Test

© 2020 Bruce Douglass

Test Report

\_ ×

- @ x

For Help, press F1

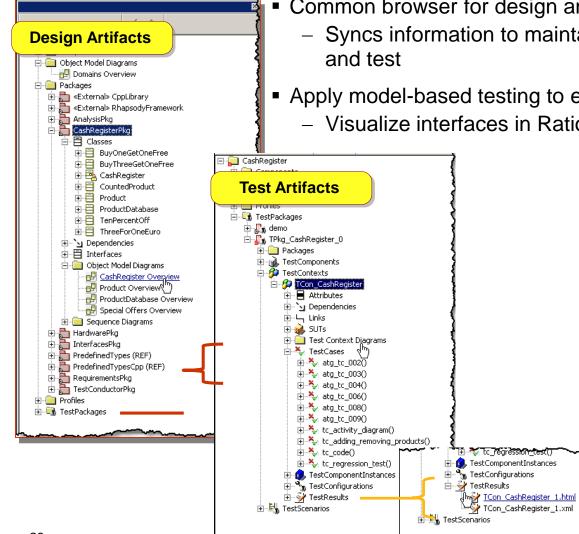
) 🛛 🖗 🖕 Select Entire Model View 👻 🗍 🛊 kenizer.itsDumm...DummyDriver\_of\_Tokenizer TCon Tokenizer.itsTokenizer.Tokenizer Status 3 Stamp Mode Components
 Object Model Diagrams - 🖏 TPkg\_Tokenize PASSED Diagram Tools 🗄 🛅 TCon\_Tokenizer\_Archite... 🙆 PASSED + Lifeline - D TCon Tokenizer PASSED System Border No.tc\_0 PASSED **«Lifeline»** Message E-S TestPackages B SD\_tc\_0 PASSED TCon Tokenizer.itsTokenizer 🖌 Event Message TCon\_Tokenizer Reply Message - 🎸 SD\_tc\_Test\_case\_1 🛛 🔘 PASSED .itsDummyDrive :Tokenizer ---> Create Message B SD\_tc\_Test\_case\_1 O PASSED Objects r of Tokenizer: ->× Destroy Message TestPackage - 🏷 SD\_tc\_Test\_case\_2 👩 PASSED Timeout TCon\_Tokenizer\_Architectur No SD\_tc\_Test\_case\_2 O PASSED DummyDriver o Dependencies TestComponents DestContexts Cancelled Time - 🍢 SD\_tc\_Test\_case\_3 🛛 🔘 PASSED 10 Time Interval N SD\_tc\_Test\_case\_3 O PASSED evDot() 🖗 TCon Tokenize - Attributes + Partition Line Comments Condition Mar evDigit('7'=\*) Dependencie Execution Occur DriverOperati Interaction Occur Interaction Opera evDigit('8'=\*) Statechart Operand Separat 🐳 SUTs - Lost Message evWS() । 😕 Tags । 🐴 Test Context Diagrams -> Found Message X Destruction Even TestCases SD\_tc\_00 SD\_tc\_Test\_case\_10 print("Result}=\*, 0.78=\*) Action Block Dependency SD\_tc\_Test\_case\_20 TestAction TestAssignment TestCondition Dependencies SDInstances TestObjectives abs(result - 0.78) < 0.001 Test/cenarios
 Test case 2 SD\_tc\_Test\_case\_30 Dependencies SDInstances TestObjectives TestResults TestScenarios Test case 3 estComponentinstance TestConfigurations TestResults TCon\_Tokenizer\_TestControl estResults 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. result Labels Off Wed, 2, Aug 2017 8:14 AM

😔 🚱 🙆 🔹 👘 🗞 🛷 😏 🔅

🖉 👫 Test case 3 in TPkg\_Toke... 🗙 💾 Test case 2 in TPkg\_Tokenize... 🖓 TC Test case 1 in TPkg\_Token... 🕥 Welcome to Rhapsody

× · · · × ■

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

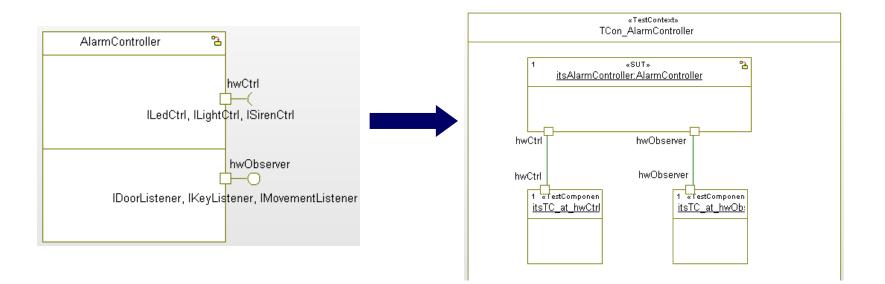


- Common browser for design and test information
  - Syncs information to maintain consistency between design
- Apply model-based testing to external code
  - Visualize interfaces in Rational Rhapsody

Test Co	ntext Result	
Test Executi	on Reports	
En	vironment Info	
Test executed on machine:	NBOSC-21-1	
Test executed by user:	ubrockmeyer	
Used OS version:	Windows 2000 / Window	rs XP
Used Rhapsody version:	Aries, build 799102	
Used TestConductor version:	2.0, build 530	
т	ested Project	
Project:	CashRegister	
Active Component:	TCon_CashRegister_5	
Active Configuration:	DefaultConfig	
Test Context: TCon_Cas	Register Summa	iry: PASSEE
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	
atq_tc_004	PASSED	

#### Automate quality

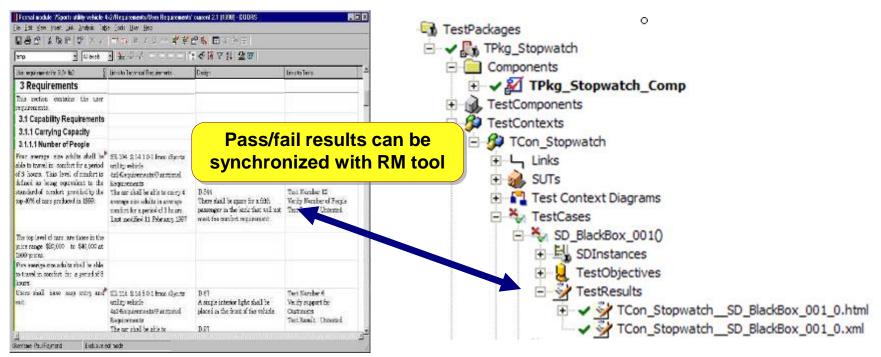
- Automatically create test architecture
  - Creates a System Under Test (SUT), test components and test context
- Apply model-based testing to external code
  - Code is developed outside of Rational Rhapsody
  - Visualize code interfaces in Rational Rhapsody and apply model-based testing



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



#### Requirements to test results coverage

• A	utomate	d reporting	of test re	esults		1	To: Requirement	-									<u> </u>	
_	- Requi	rement to te	st cova	rade table	<b>`</b>			REQ1	REQ2	REQ3	REQ4	REQ5	REQ6	REQ7	REQ8	REQ9		
	•			lage lable	,		SD_tc_0 30_tc_0 30_tc_007										REQ10	
-	- Test C	Coverage re	sults			SICa	atg_tc_007											
-	- Comp	lete test res	ults in R	Rational			🔖 atg_tc_002											
		hing Engine				Scop	🍇 atg_tc_003	🔒 REQ1										
				•			🐳 atg_tc_004											
							🐴 atg_tc_016	A 11 D										
							∛y atg_tc_017	All K	lequire	ments	\$							
E Ra	tional Quality M	anager						Nam	e Speci	fication								Covered by Test Case
	Home View Test P	ans 📋 TestPlan CashRegist	e 📔 TestCase (	01 SD InitC	cution Result			REQ1				sh Regist 1er has s		to be de	signed th	nat reads	s barcodes of	atg_tc_003 ( <b></b> Passed)
					auon nesun			REQ1	0 After r display		a start i	event Ca	sh Regist	er will se	end a mes	sage "sł	how(Ready)" to its	SD_tc_0 ( <b>F</b> ailed)
E Planning	Command Lir	n Result 🤔 ne Result	-	. Const Donald				REQ2	If the								ved on a display. hknown product"	not covered
E				t Case Result est Case: SD_tc_0				REQ3	will be	displaye	d and th	e barcod	e can be	entered	via the (	ashier's	keyboard.	not covered Code_tc_0
onstruction			10:20:31	1, Monday, April 27, 2009 Environment Info	Name			RF04	wnen	all the se	elected p		verdic				ted containing the otal price.	( <b>E</b> Passed)
	Actual Result: Host Name:	Passed jekyllsslave	Test executed on machine: Test executed by user:	JEKYLLSSLAVE Administrator		_CashReg	ster SD	) to (	0 4.ht	ml			🐔 Fa			ole to ad Euro".	ld special offers	not covered
Lab lanagement	Owner: Test Milestone:	Mary, Test Manager	Used OS version: Used Rhapsody version: Used TestConductor version:	Windows 2000 / Windows XP 7.5, build 1155117 2.4, build 1406		_CashReg								assed	1			FC_tc_0 ( Passed)
Execution	Test Case: Test Script	TestCase_01_SD_InitCashR SD_tc_0	Project:	Tested Project CppCashRegister	💕 TCon	_CashReg	sterat	g_tc_	006_9	9.html		(	🛐 Pa	assed	el ti	ne last s	elected product,	not covered
	Test Data:	Unassigned	Active Component: Active Configuration:	TPkg_CashRegister_Comp DefaultConfig	📝 TCon	_CashReg	ster_at	g_tc_	002_9	9.html		(	🛐 Pa	assed	forr	ns in the s	e future.	not covered not covered
Reports	Weight:	100	TPkg_CashRegister::SDTestSce	SDs used in test mario_0	💕 TCon	_CashReg	ster_at	g_tc_	003_9	).html		(	🛐 Pa	assed				not conorca
esult Det!			Summary Total number of SDs used:	Info Summary: pa	😼 TCon	_CashReg	ster_at	g_tc_	004_9	).html		(	r 🛐 Pa	assed				
estConducto	registerSD_tc_0_0.html brAdapter20844.out brAdapter20845.err		Total number of SD instances in Total number of executed SD in Total number of PASSED SD ins	istances: 1	😼 TCon	_CashReg	sterFC	_tc_(	0_0.ht	:ml		(	r 🛐 Pa	assed	1			
estLog2084			Total number of FAILED SD inst Total number of ACTIVE SD inst Total number of NOT ACTIVE SI	tances: 0 ( 0%) tances: 0 ( 0%)	😼 TCon	_CashReg	ster_Co	ode_to	c_0_0	.html		(	S Pa	assed				
			Total number of NOT ACTIVE S	o realizes: 0 (0%)	💕 TCon	_CashReg	ster_7.h	tml				(	🛐 Fa	ailed	T			

#### Coverage Analysis is one of the key benefits of automation

Which requirements are covered?

1	E	ReqCoverage X										
	1	fo: Requirement Scope: Cpp	CashRegist	er								
l <sub>T</sub>			() REQ1	[] REQ2	<ol> <li>REQ3</li> </ol>	[] REQ4	() REQ5	() REQ6	[] REQ7	[] REQ8	C REQ9	🚺 REQO
rom:	X	TestCase_simple_start										📙 REQO
르	×	TestCase_code_assert										
es.	X	TestCase_Flow_Chart										
estCase	×	Code_tc_0						🗎 REQ6				
BS	Ň	SD_tc_0										

Det	ailed Coverage Summary	y of CashRegister (9/25)
Operations		
not covered	identifyProduct	
covered	addProduct	
covered	startSession	
not covered	endSession	
not covered	<u>generateTicket</u>	
covered	isNoMoreProducts	
not covered	removeLastProduct	
covered	countProducts	
EventRecepti	ons	
covered	<u>evStart</u>	N
not covered	<u>evBarcode</u>	Click to highlight element in Rhaj
not covered	<u>evEnd</u>	Ciex to highlight element in Kha

Which model elements are covered?

#### 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						

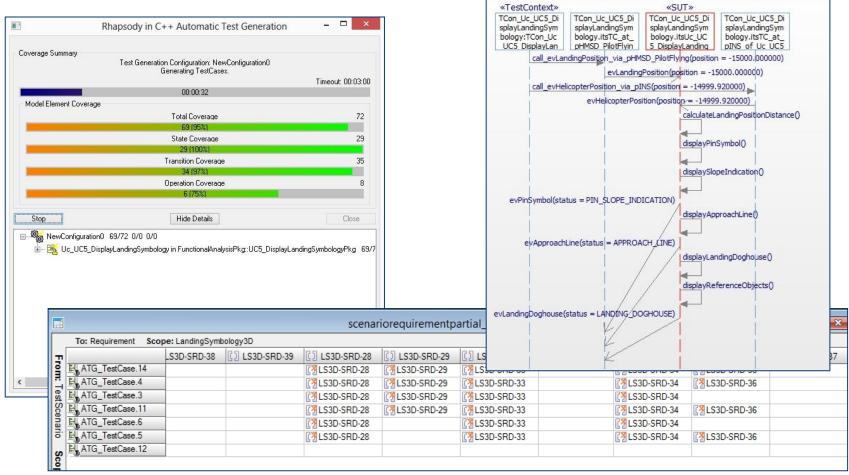
	dodia	001010	-u
Statement Coverage	70	43	61.4%
Decision Coverage	6	1	16.7%
Condition Coverage	0	0	n.a.
Condition/Decision Coverage	20	7	35%
Modified Condition/Decision Coverage	20	7	35%

Coverage Report

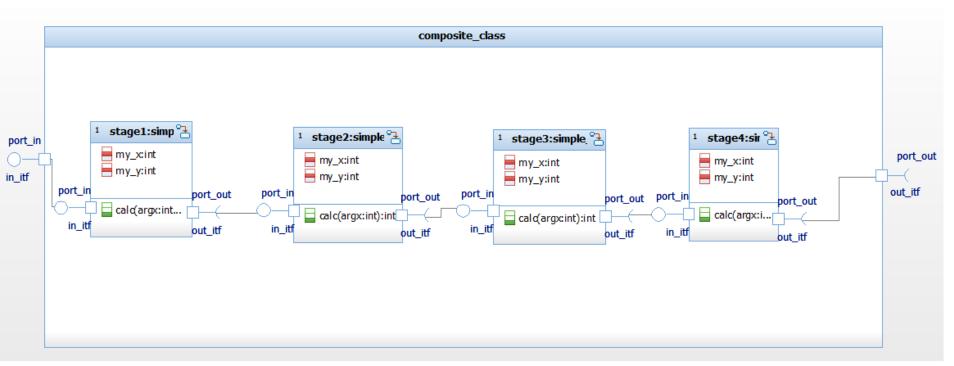
Envir	onment Ir	fo Table Of Contents	Global Statistics	Source Code
	33b		1	
	34	cleanUpRelations();	·	
	35 }			
	36			
->	37 bo	ol CashRegister::hw_C::InBound_C::send(	IOxfEvent* event, const IOxfEv	entGenerationParams& params
	37b			
	38	hool res = false;		
т?	39	<pre>if (event != (0))</pre>		
	39b	(		
	40	event->setPort(getPort());		
T ?	41	<pre>if (itsIBarcodeReader != (0))</pre>		
	41b	(		
? F	42	<pre>if (event-&gt;isTypeOf(24601))</pre>		
	42b	{		
	43	res = itsIBarcodeReader->	send(event, params);	
	44	return res;		
	4.5	)		
	46	)		
T 7	47	if (itsIKeyboard != (0))		
	47b	}		
7 1	48	if (event->isTypeOf(24602))		
	48b			
	49	res = itsIKeyboard->send(	event, params);	
	51	Letain Les;		
	51	) if (event->isTypeOf(24604))		
	52b	rr (evenc->isiypeor(24604))		
	53	res = itsIKeyboard->send(	erent, naraba):	
	54	return res;	evency percest;	
	55	Leona Leon		

#### MBT – Automatic Test Generation (ATG)

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

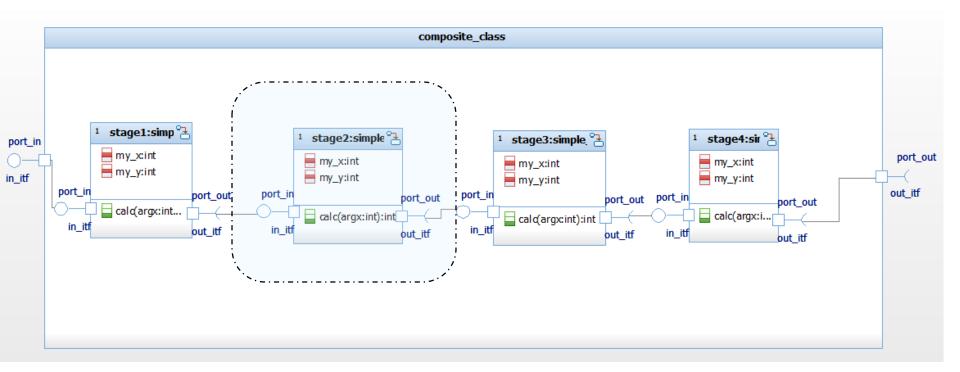


#### Sample System to demo MBT

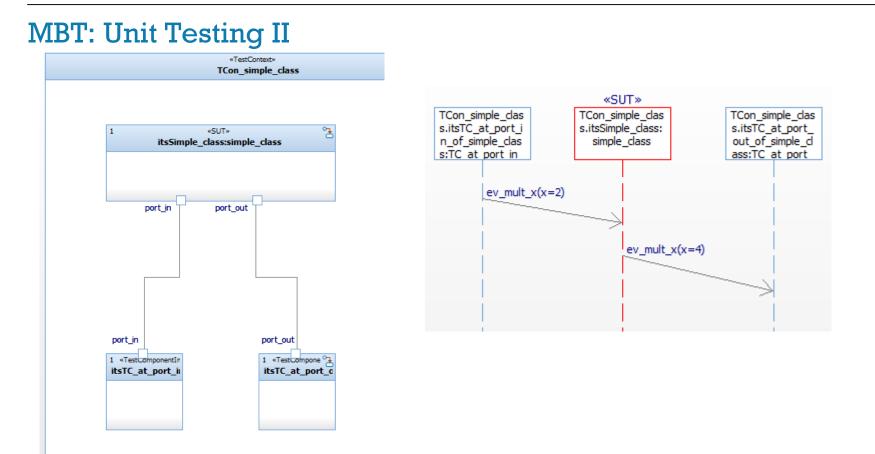


- System shows an explicitly modeled input and output interface using ports
- System contains four units with explicitly modeled input and output interfaces using ports; the units get input integer values and multiply with 2
- Software architecture shows how the units are integrated using ports and links

#### **MBT: Unit Testing I**

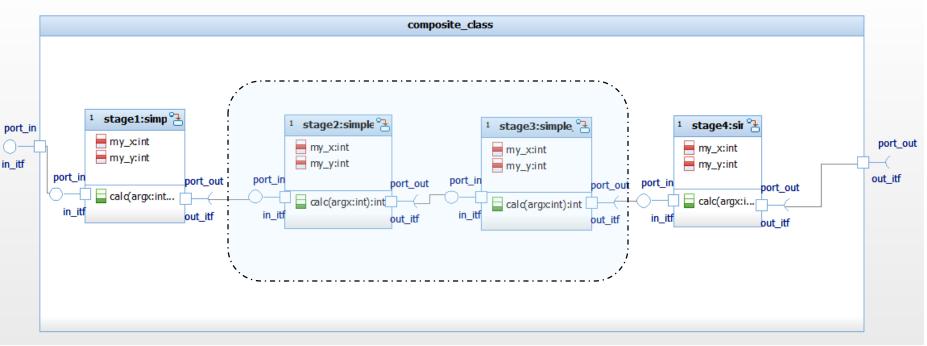


- Objective is to test each unit in isolation
- TestCondcutor automatically creates test architectures for each unit (SUT)
- "White box test":
  - requirements based testing using the interfaces of the SUT
  - code coverage measurement of the internal structure of the SUT



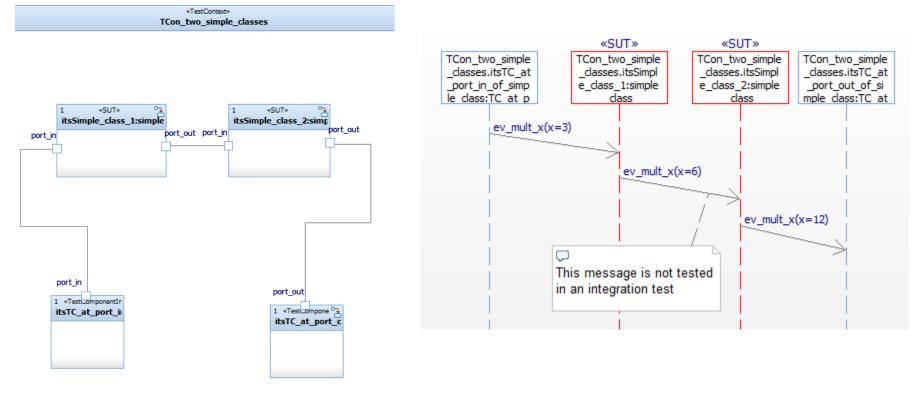
- An instance of the unit under test (SUT) is contained in the test architecture, and two test components which are connected to the ports of the SUT
- Developers specify the expected input / output behaviour in a test case
- TestConductor executes the unit tests and computes test verdicts (pass/fail)

#### **MBT: IntegrationTesting I**



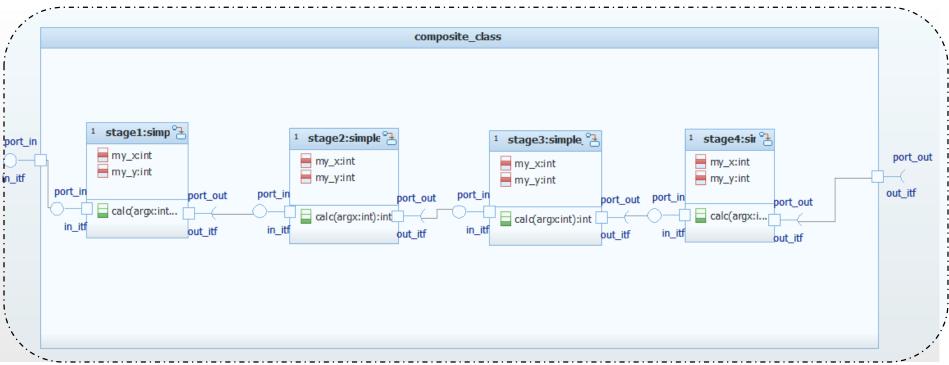
- Objective is to test two or more integrated units
- TestCondcutor automatically creates test architectures for one unit, developers can extend the test architecture to add more units (SUT)
- "Grey box test"
  - requirements based testing using the external interfaces of the integrated SUT
  - code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





- 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

