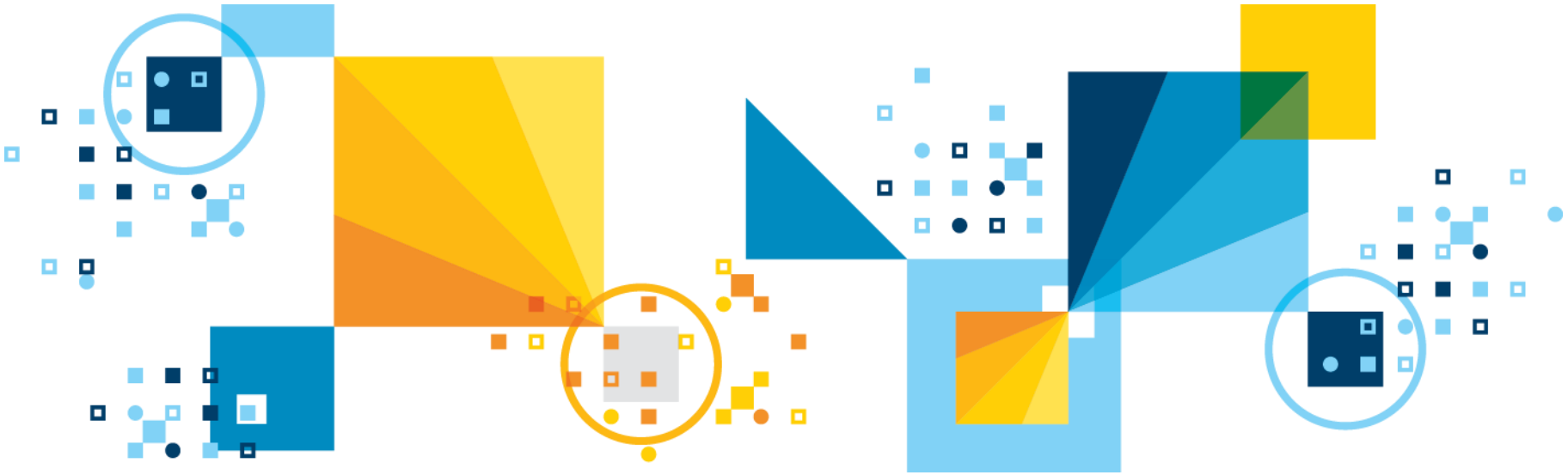
What is Model-Based Testing … and how do I get started?

Bruce Powel Douglass, Ph.D.
www.bruce-douglass.com
Bruce.Douglass@outlook.com



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

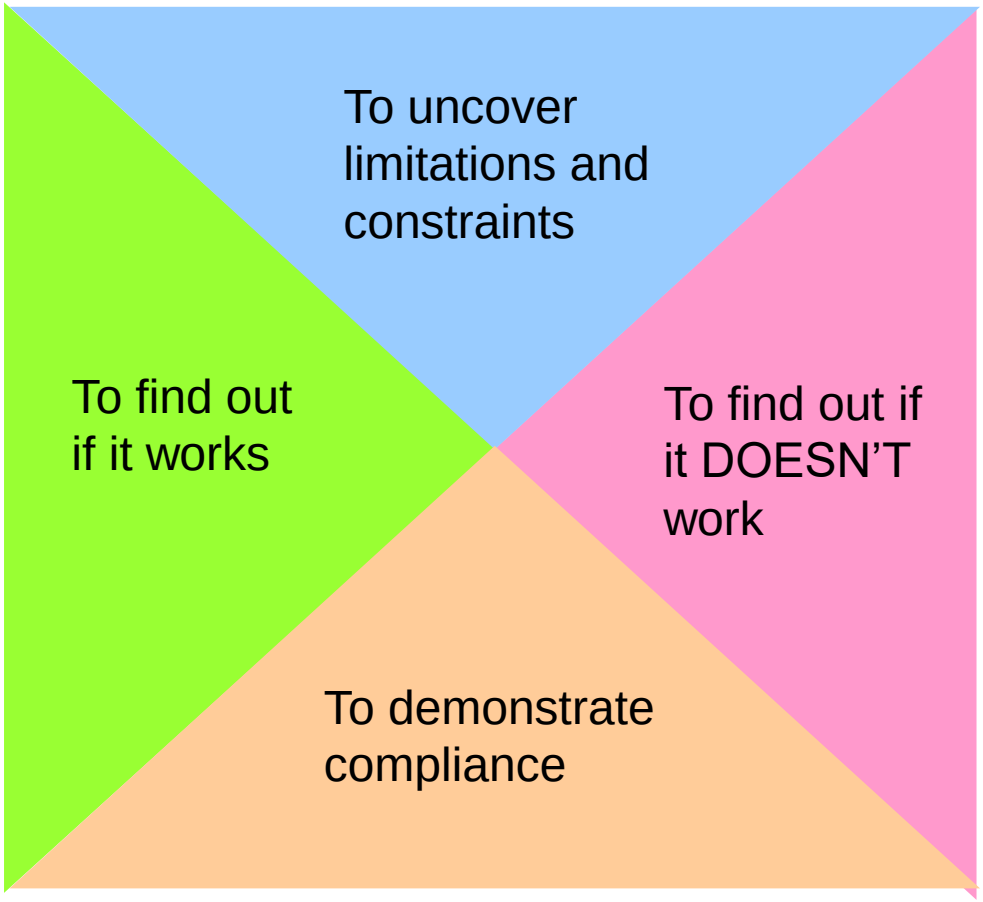
"Works"

Level 1

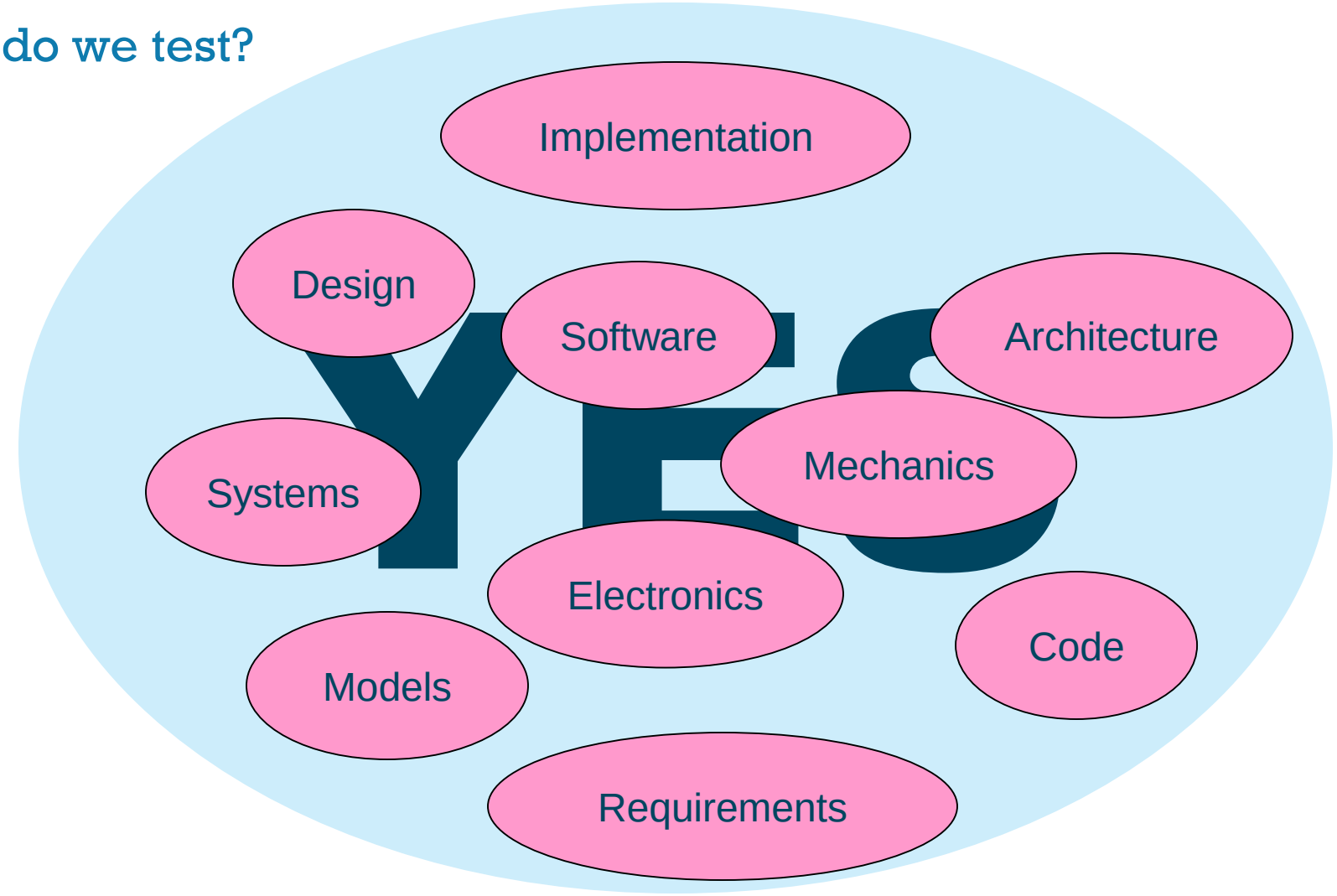
Kinda sorta

Level 0

Why do we test?



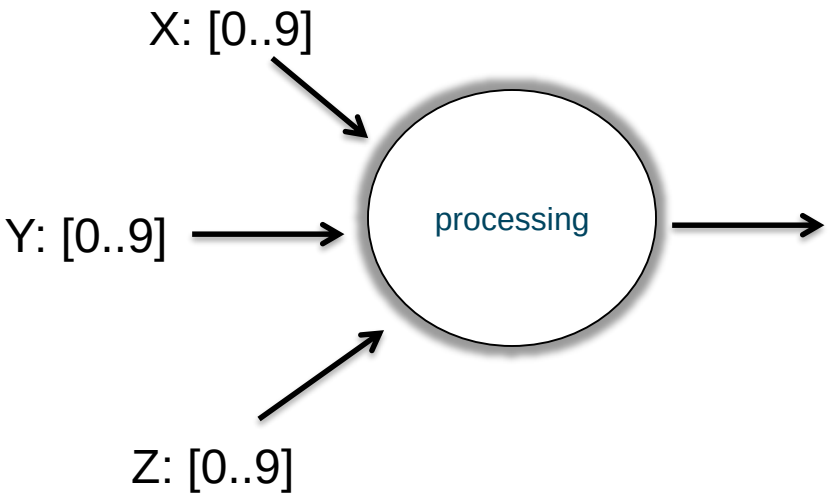
What do we test?



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

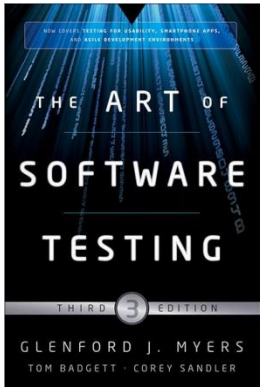


Testing can never be complete – there are an essentially infinite set of combinations of value, sequence, and timing

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

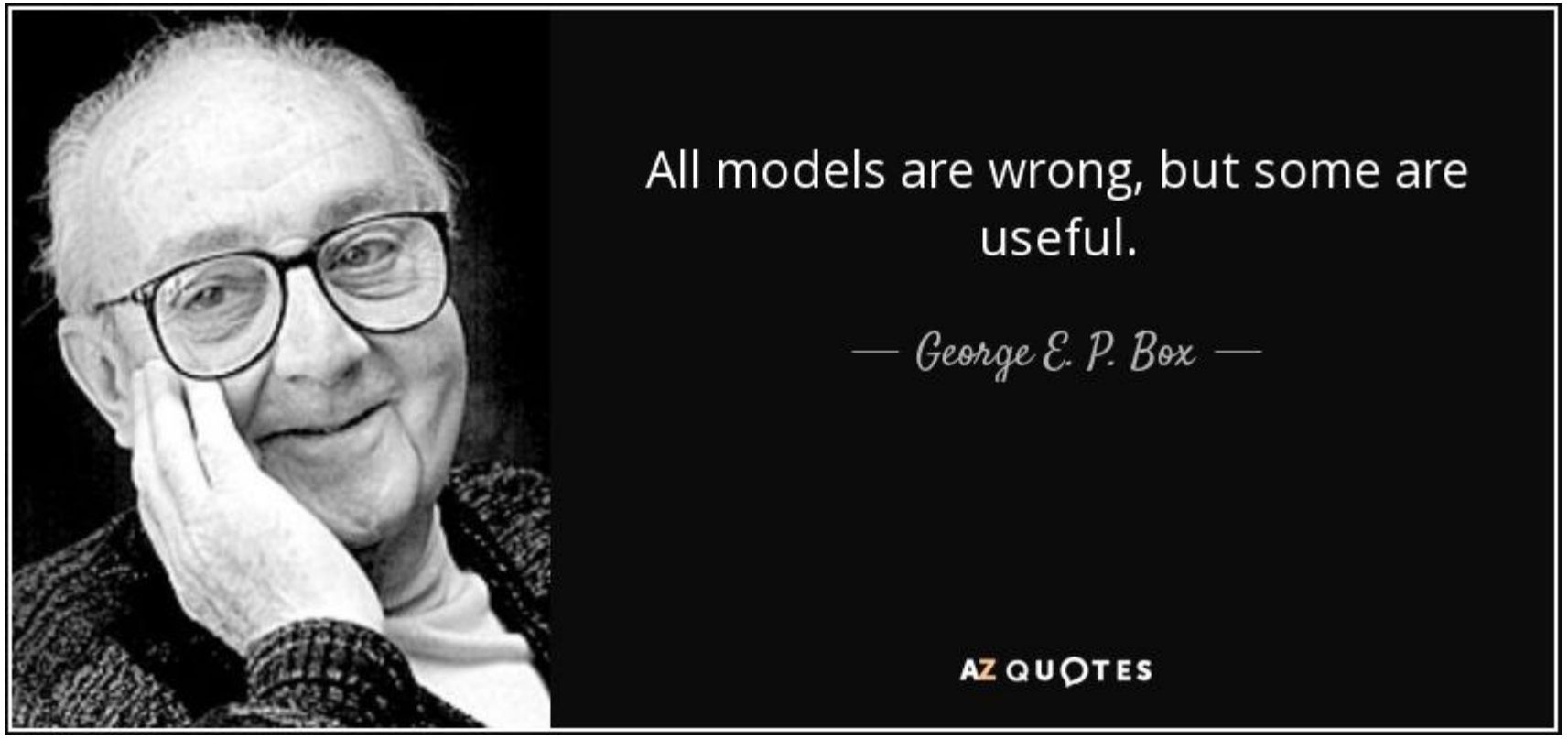
- X comes before Y? Or Z before X?
- The system expects Z to occur in < 20ms but it arrives at 30ms?
- The output comes too late?
- What if Z, Y, and Z are not independent? Example: if X>5 then Y must be <= 2
- What if X is -1?
- Does the case Z==-20 fail in the same way as X == 45?
- What if X and Y 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*



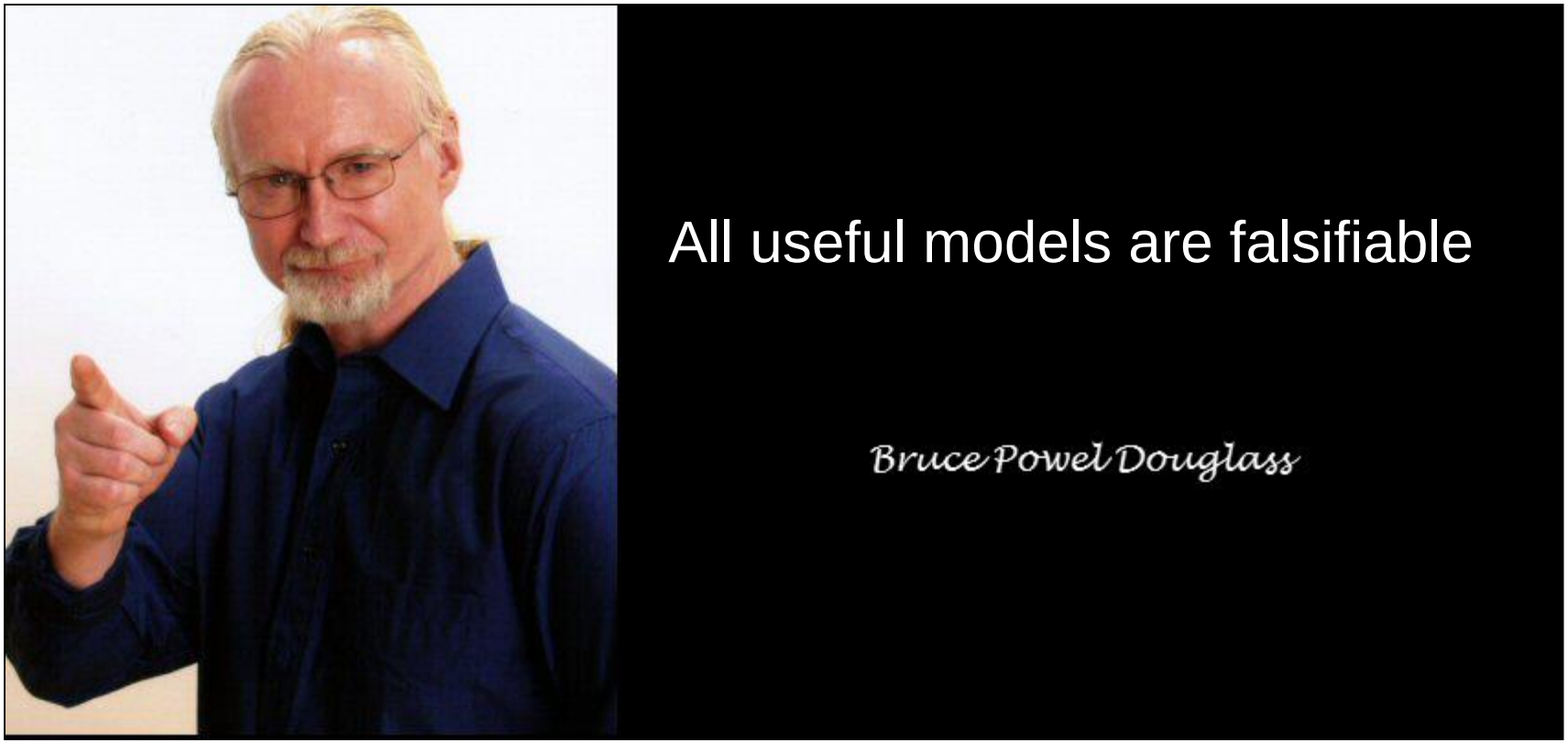
- Consider the simple problem
  - *The program reads three integer values from a text input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.*
  - **Define test cases for this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



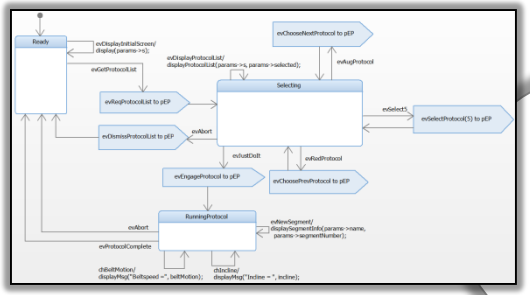
- Problem: Reality is too complex
- Solution: Create a model
- A model is always a simplification of reality, wherein we focus on aspects relevant to things we care about and elide details of those things we do not.

Models

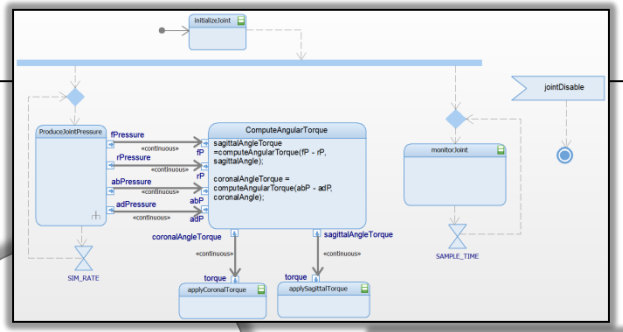


- Rigorously defined – computable – models make statements that can be demonstrated to be true or false
- A subtype of computable models – known as executable models – can be tested

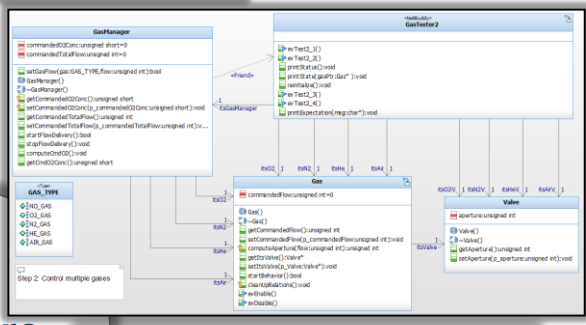
Modeling Views

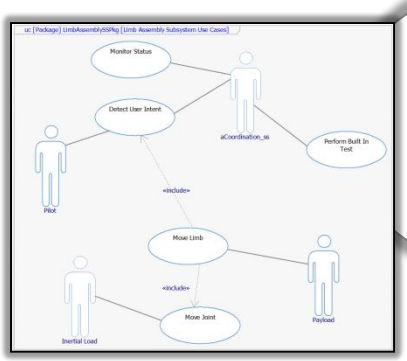


State Behavior

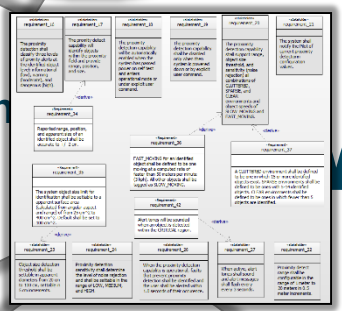


Flow Behavior

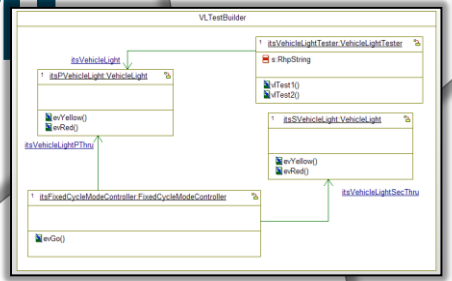


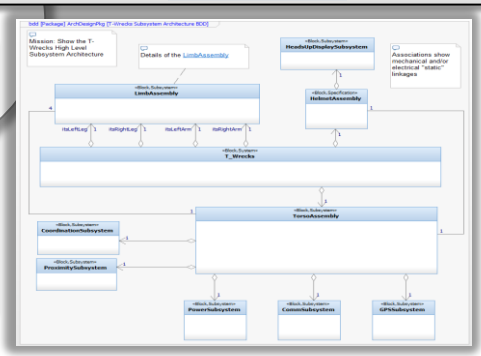


Function ality

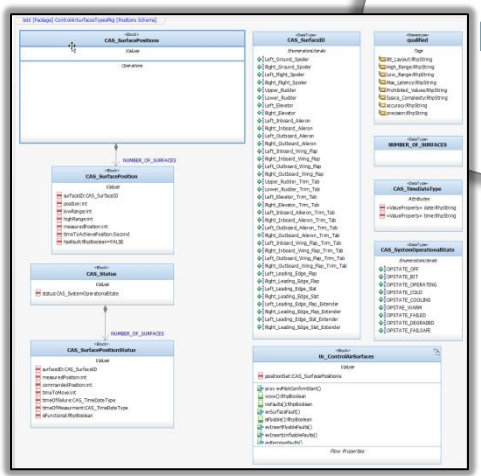


Structure

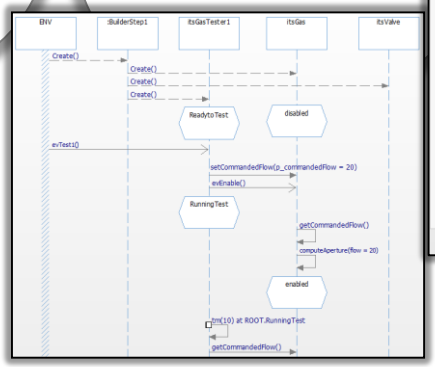


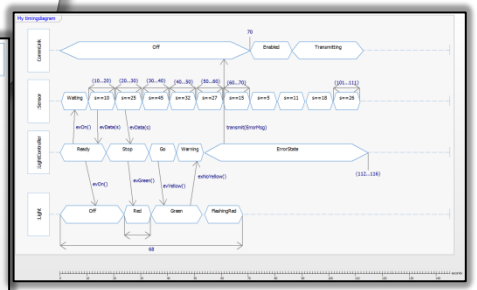


Data

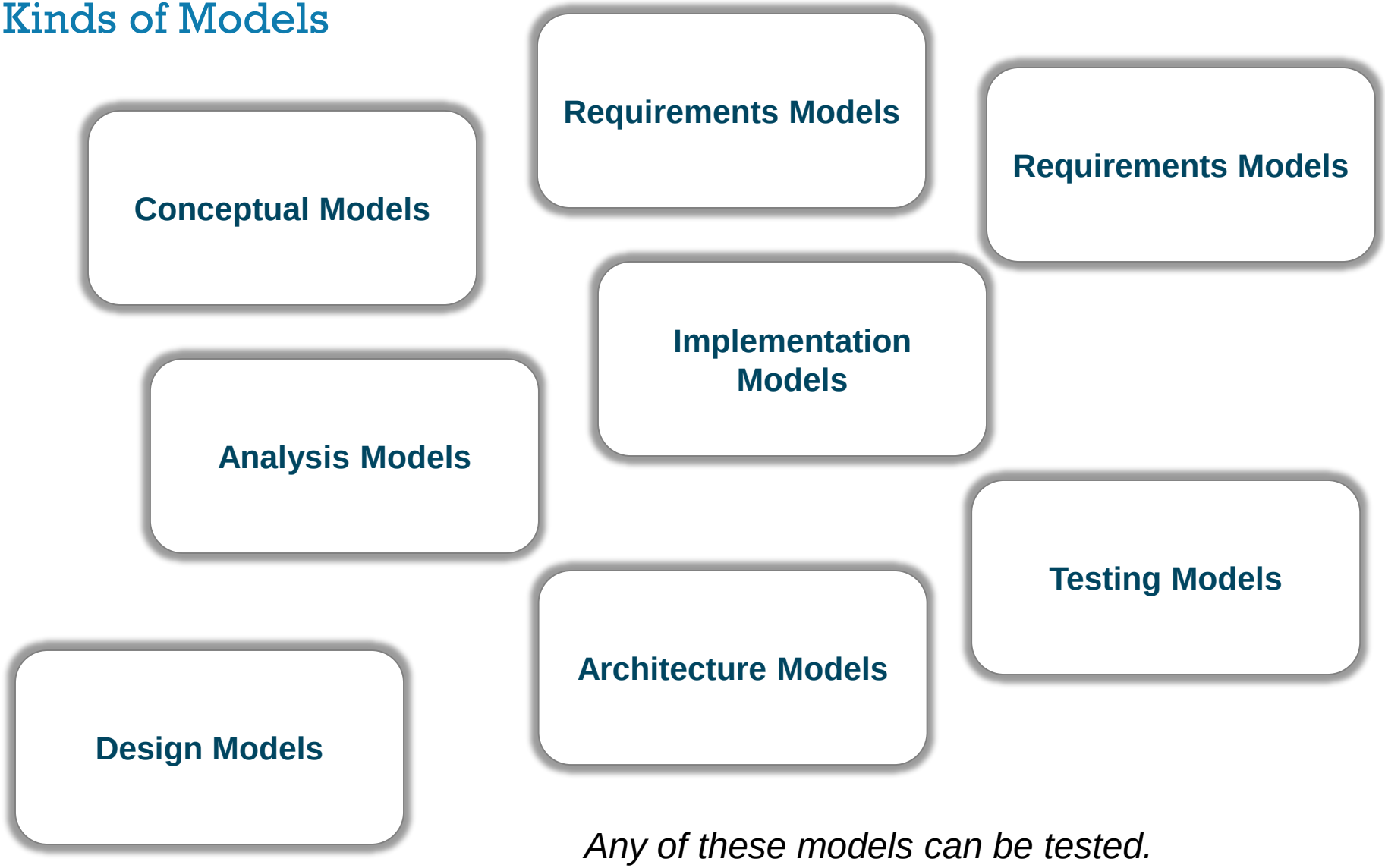


Interactions





Kinds of 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

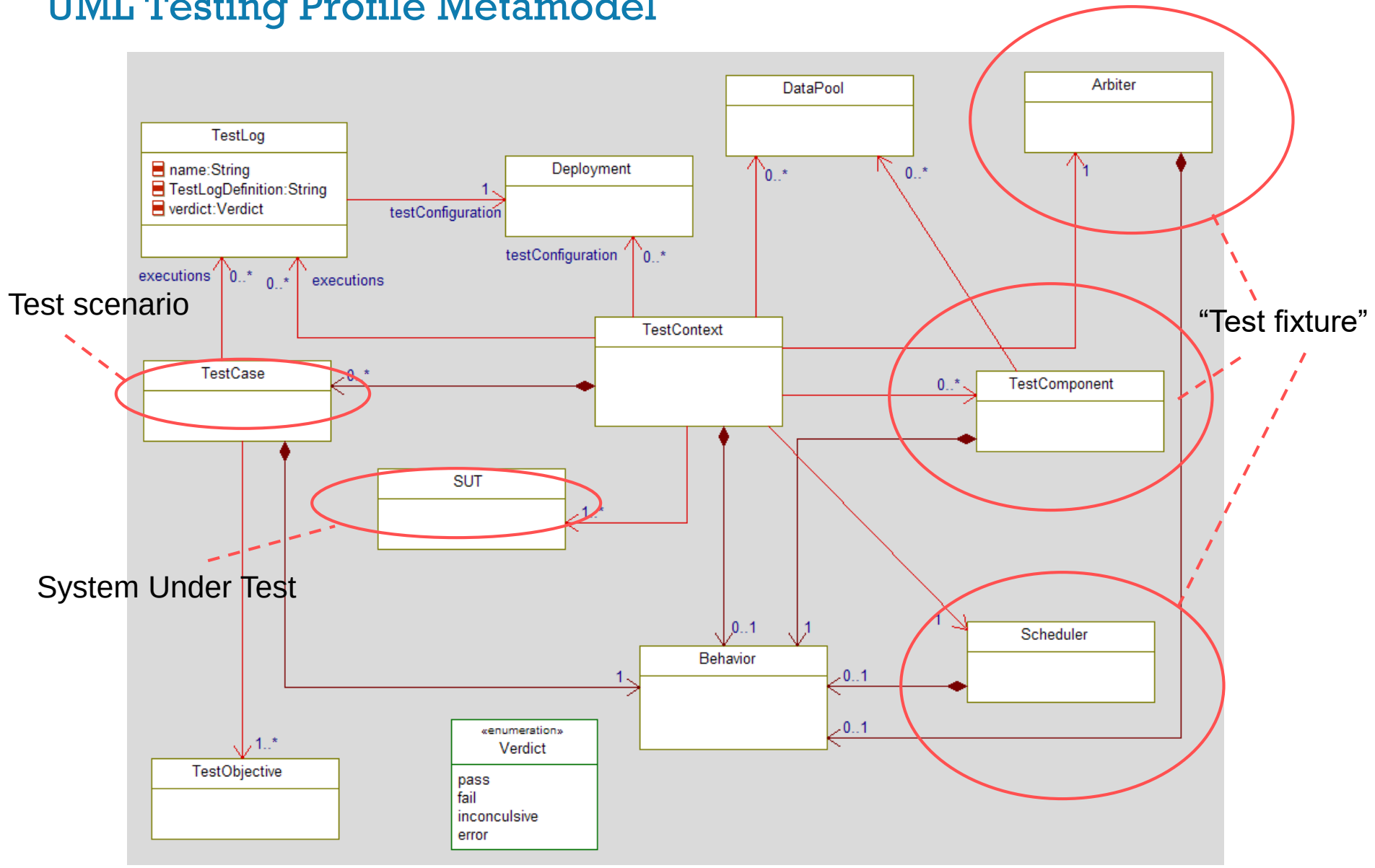
UML Testing Profile

- 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.

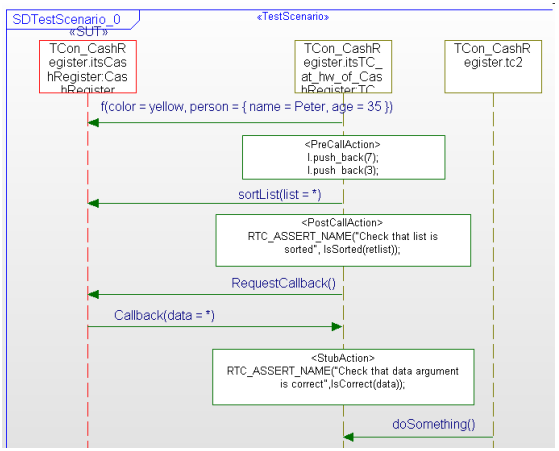
The UML Testing Profile extends UML with test specific concepts like test components, verdicts, defaults, etc. These concepts are grouped into concepts for test architecture, test data, test behavior, and time. Being a profile, the UML testing profile seamlessly integrates into UML: it is based on the 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.

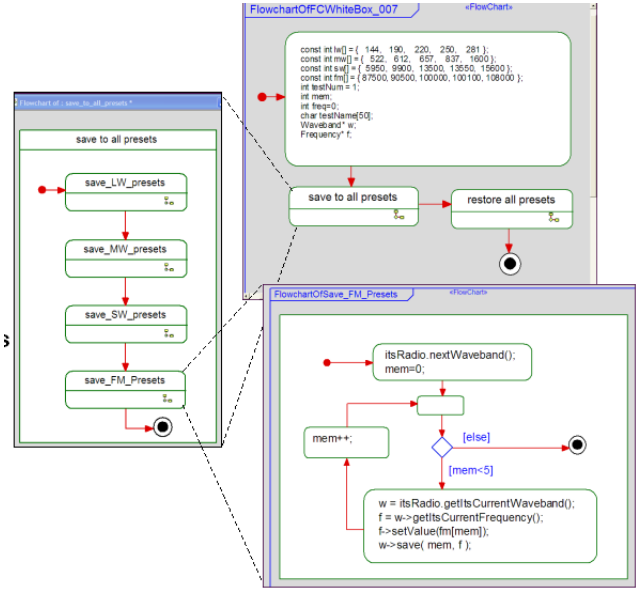
UML Testing Profile Metamodel

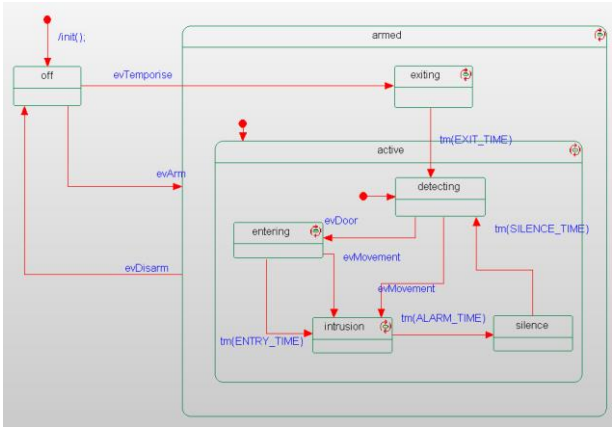


Capture test cases with UML/SysML

- 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)







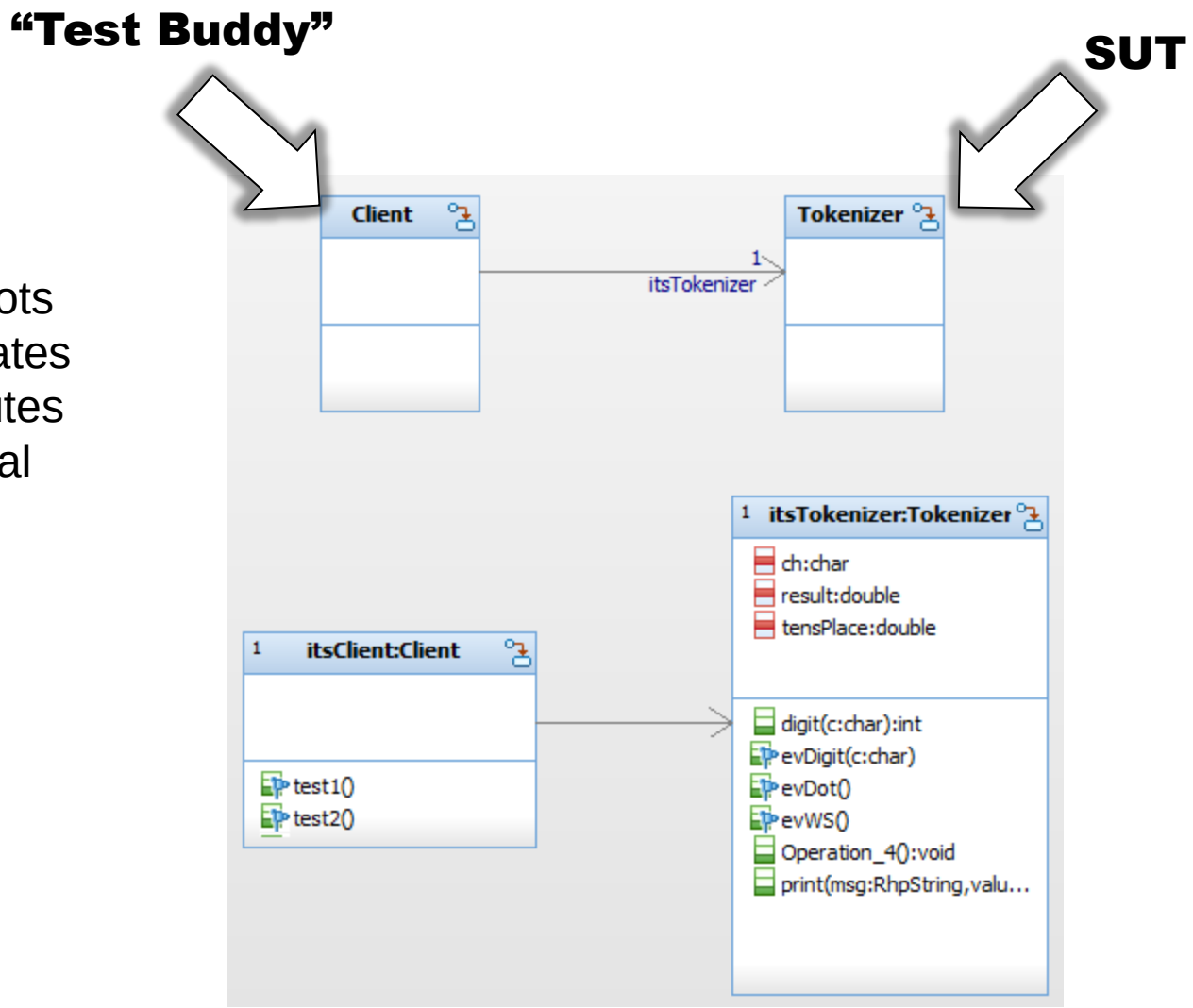
Sequence Diagram Test Case

Flow Chart Test Cases

Statechart Test Case

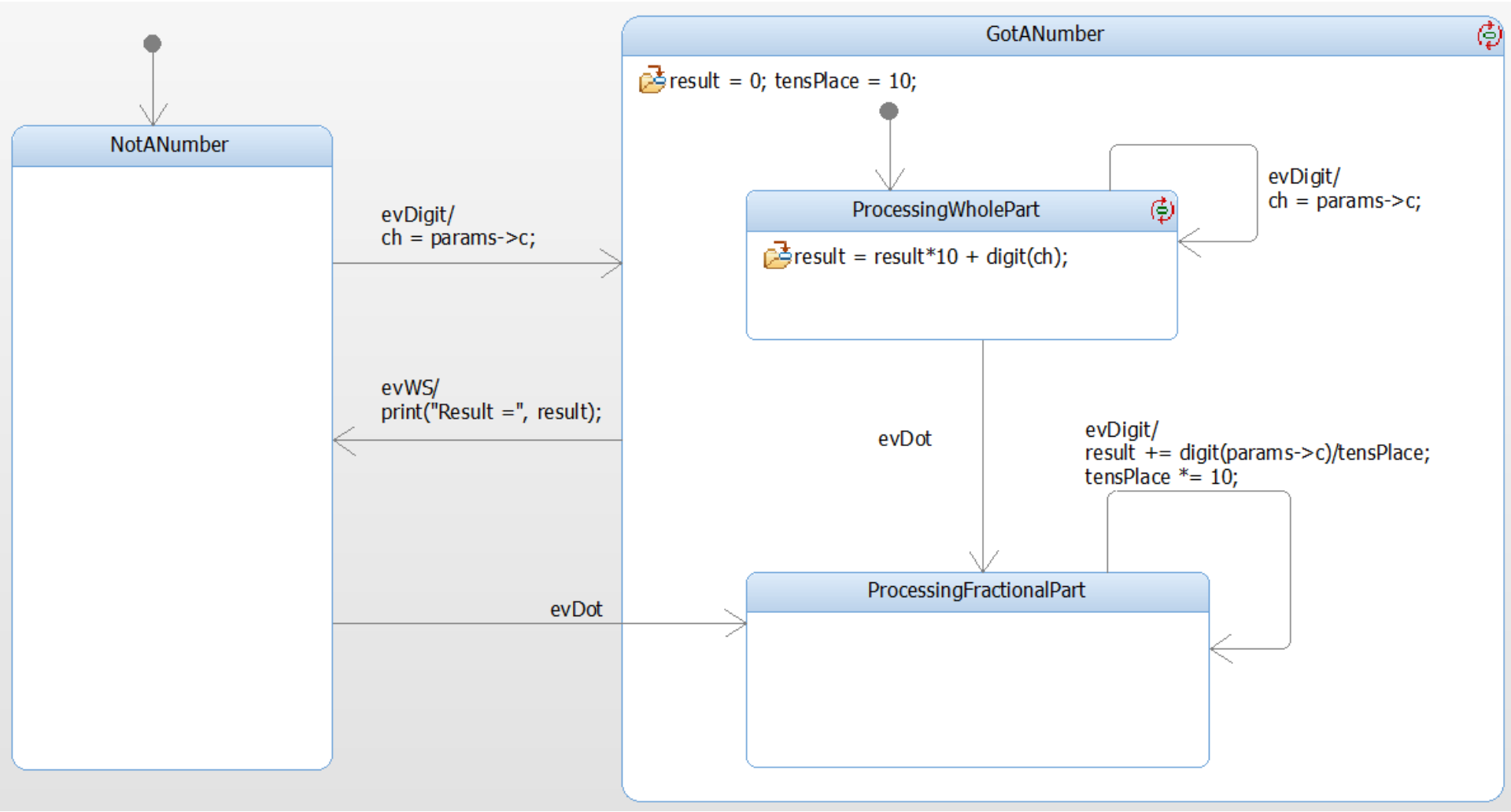
Example model: Tokenizer (Manual)

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



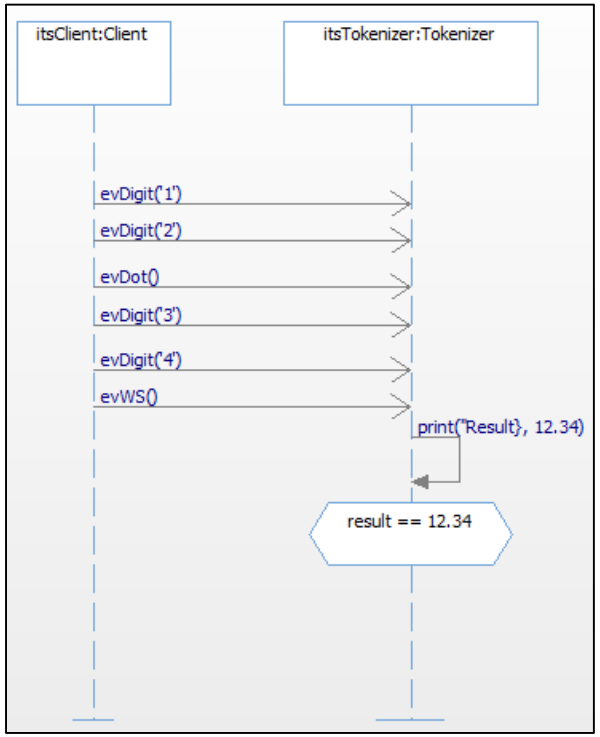
Example model: Tokenizer (Manual)

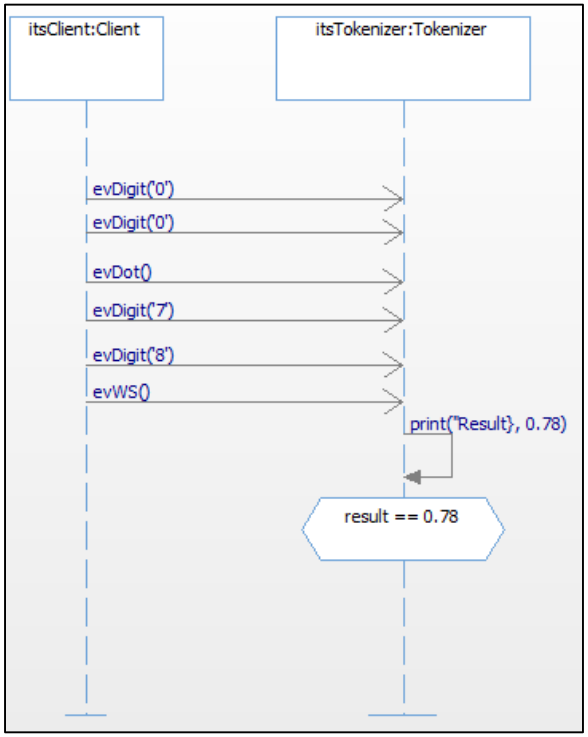
This is the state machine for the Tokenizer class

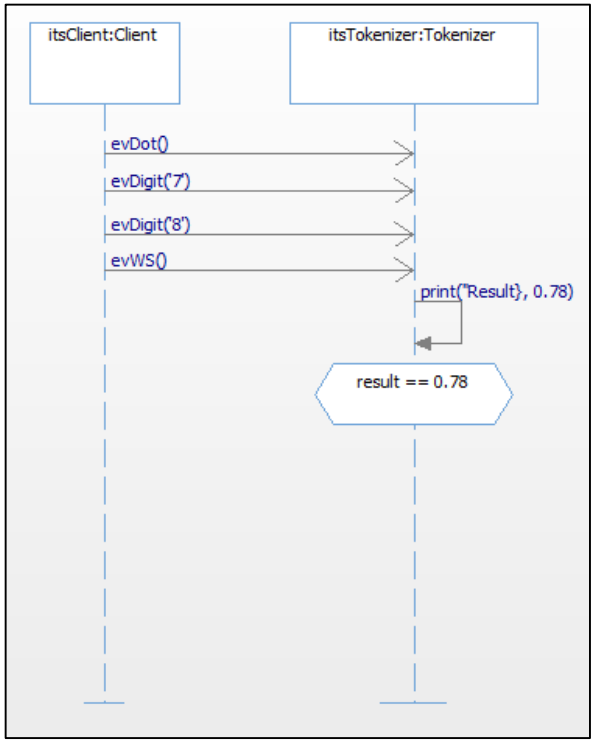


Example model: Tokenizer (Manual)

Create Test Cases as Sequence Diagrams

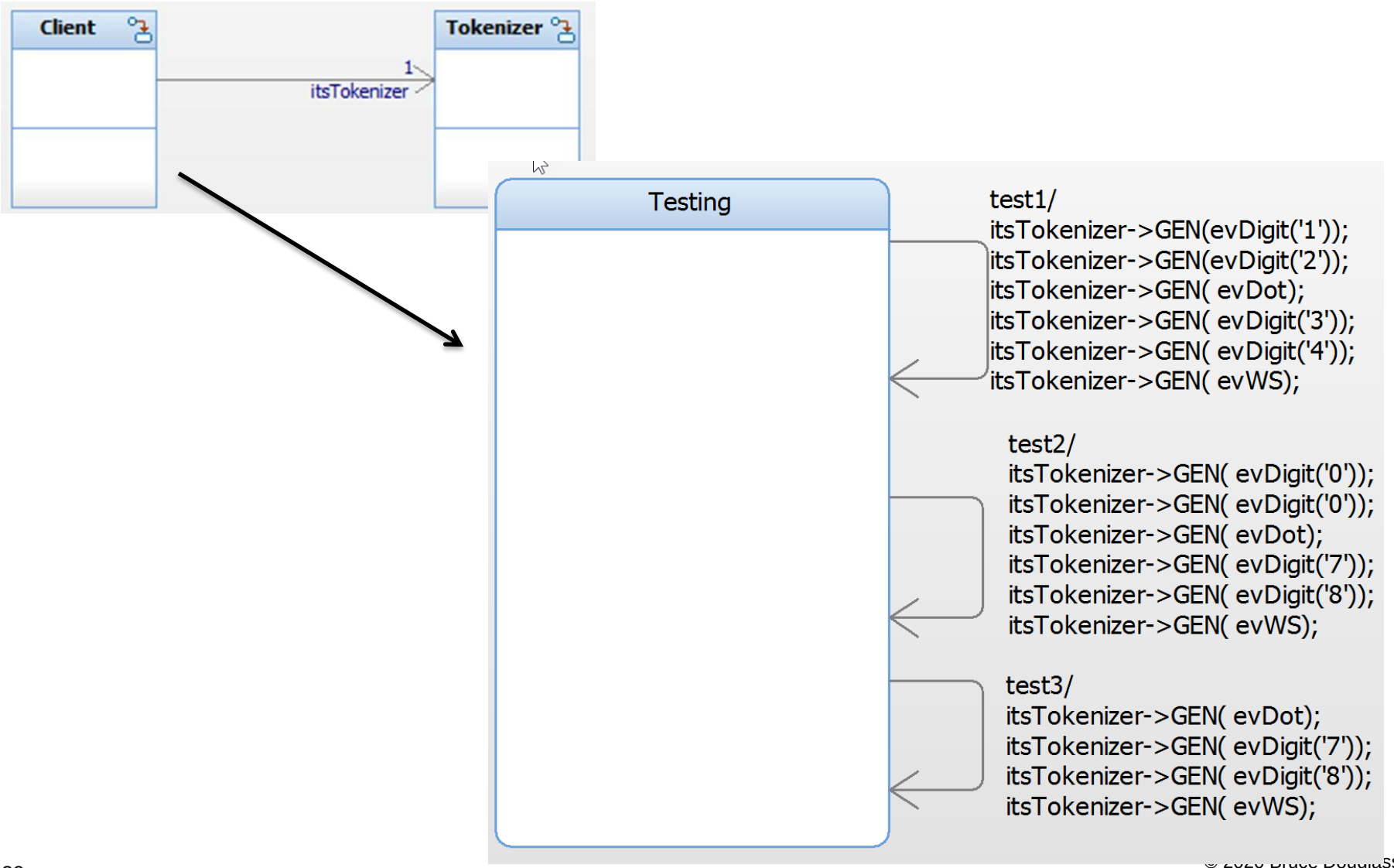






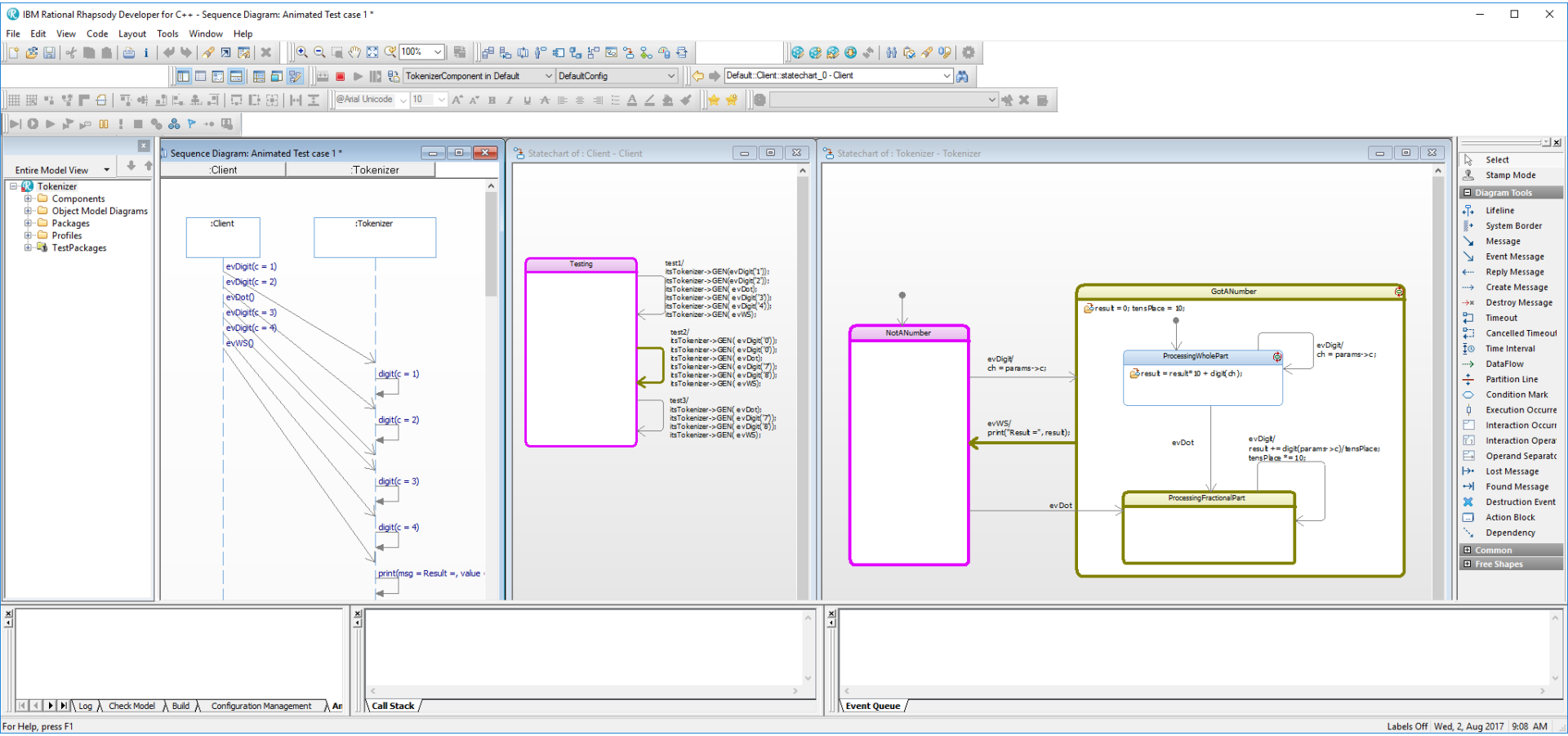
Example model: Tokenizer (Manual)

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



Example model: Tokenizer (Manual)

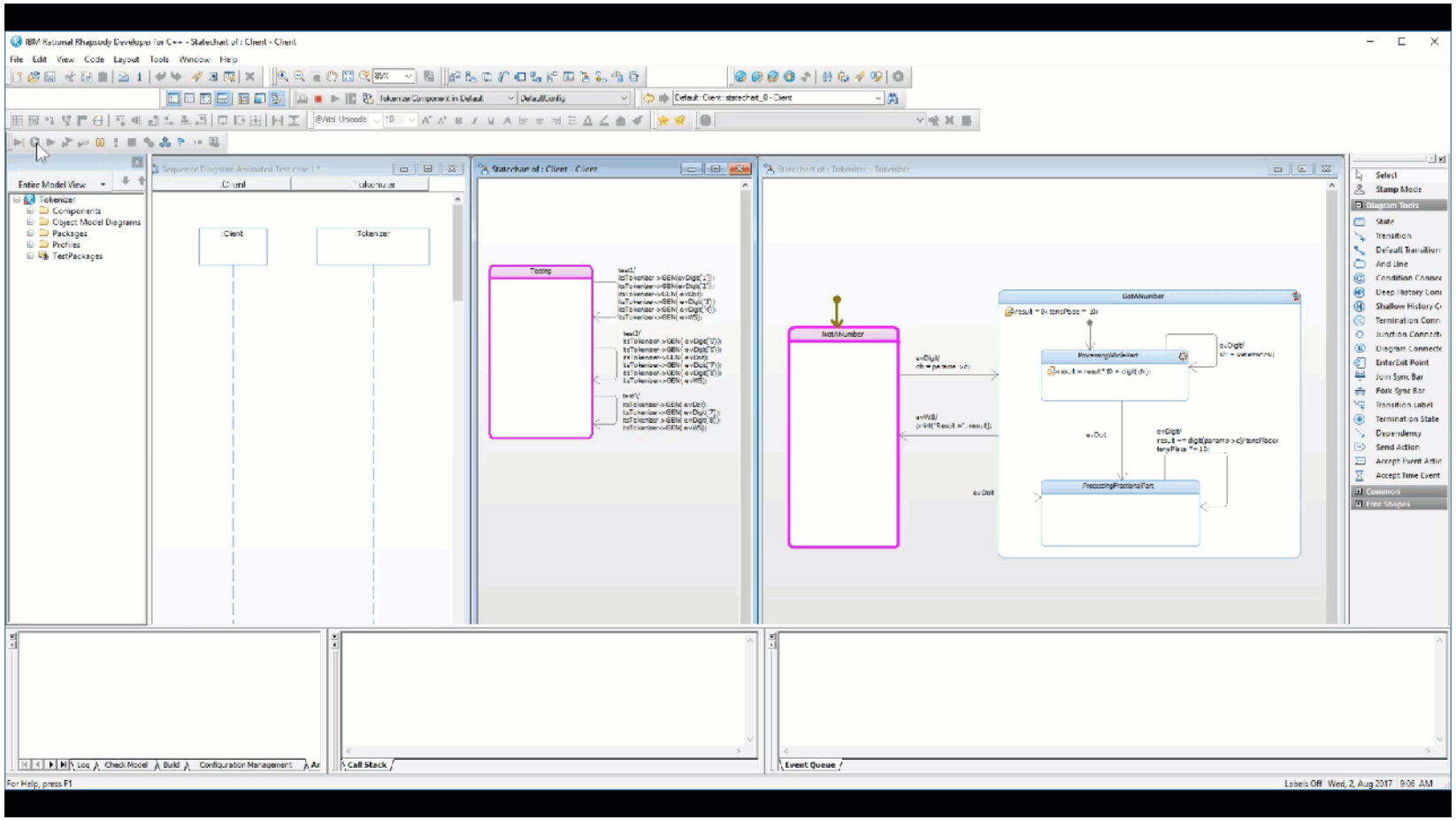
Now execute the model and create “animated sequence diagrams”* from the execution)



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

Example model: Tokenizer (Manual)

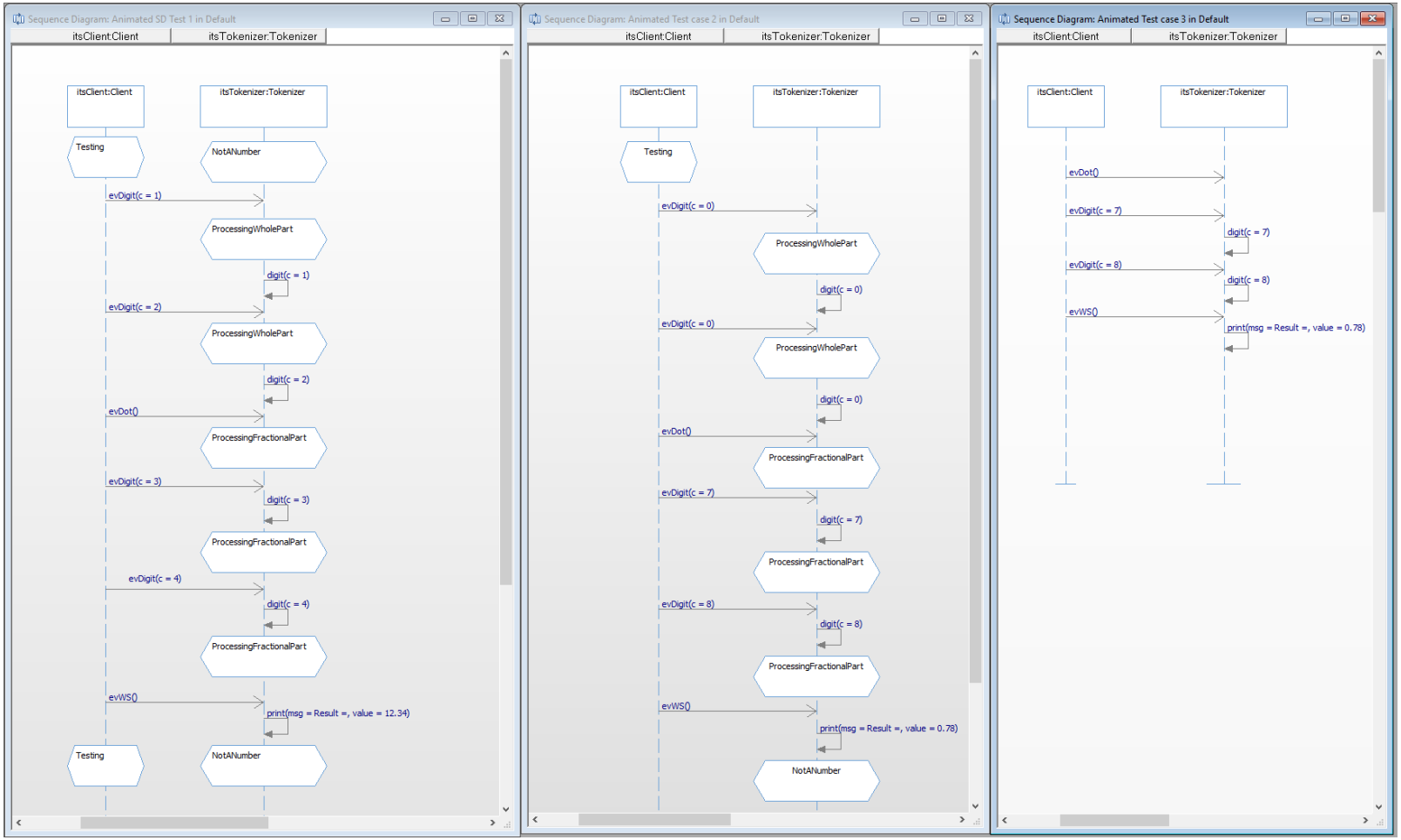
Now execute the model and create “animated sequence diagrams”* from the execution)



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

Example model: Tokenizer (Manual)

Review the outcomes and compare to the test specifications

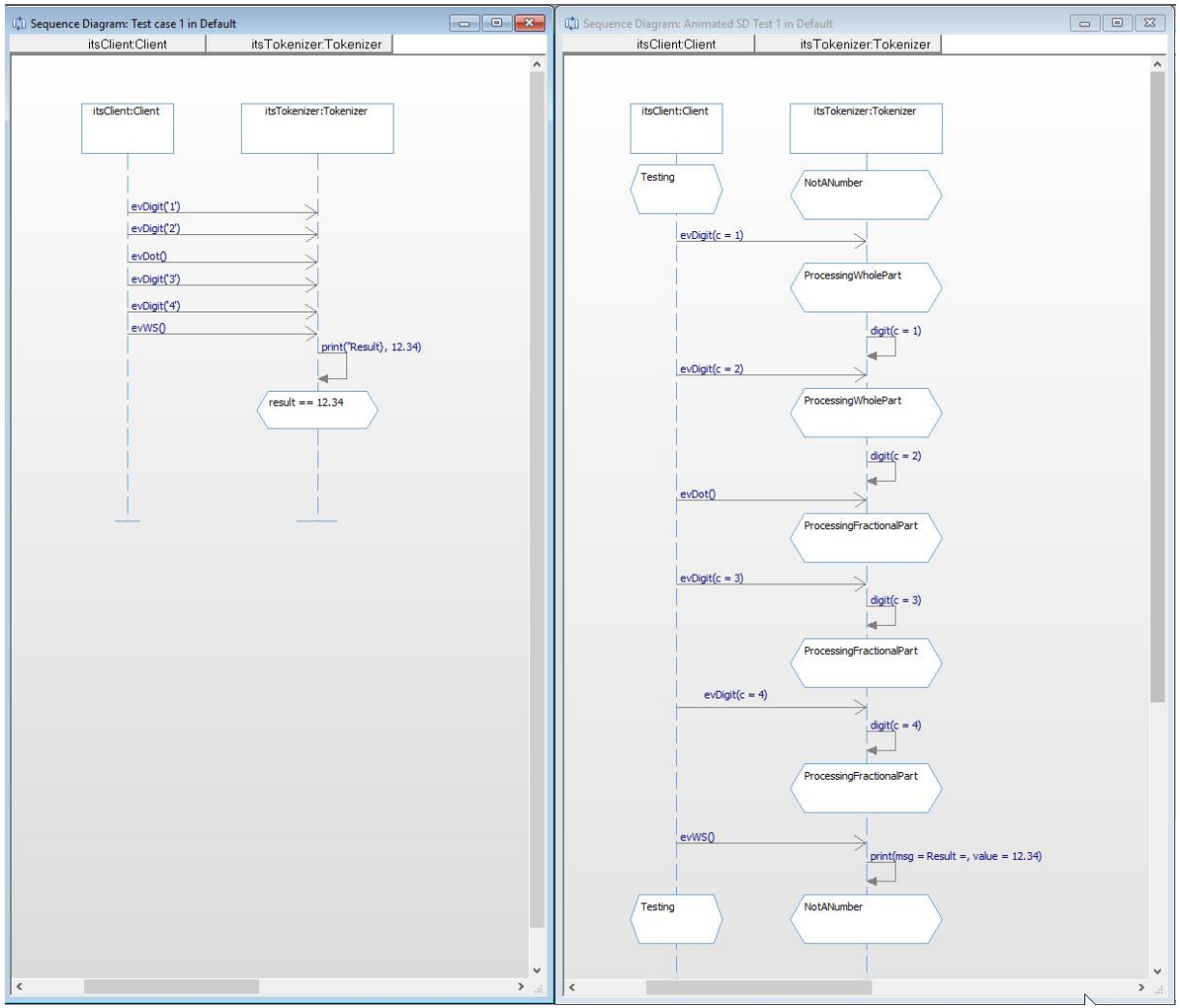


Test Case 1 Outcome Test Case 2 Outcome Test Case 3 Outcome

Example model: Tokenizer (Manual)

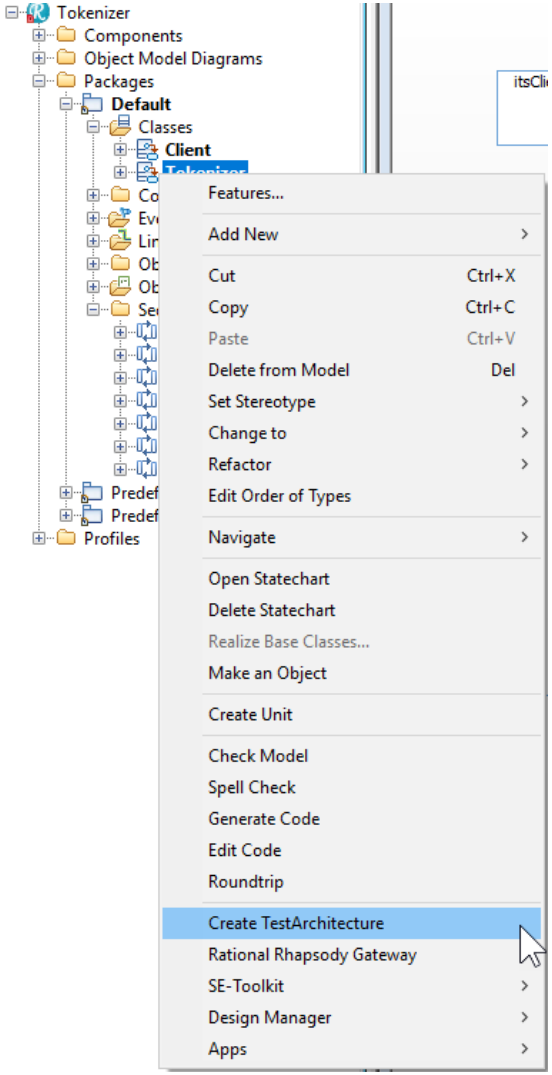
Review the outcomes and compare to the test specifications

Test Case 1

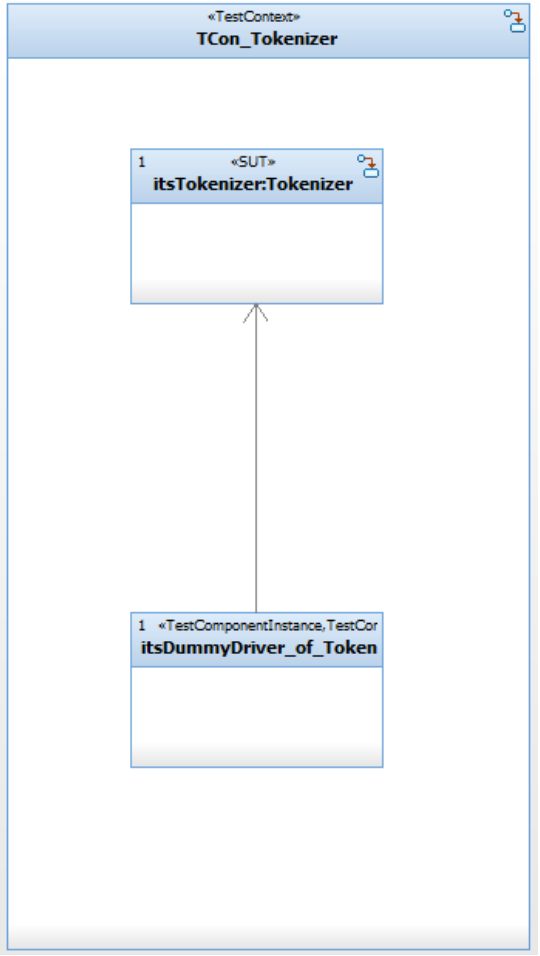


Test Case 1 Result

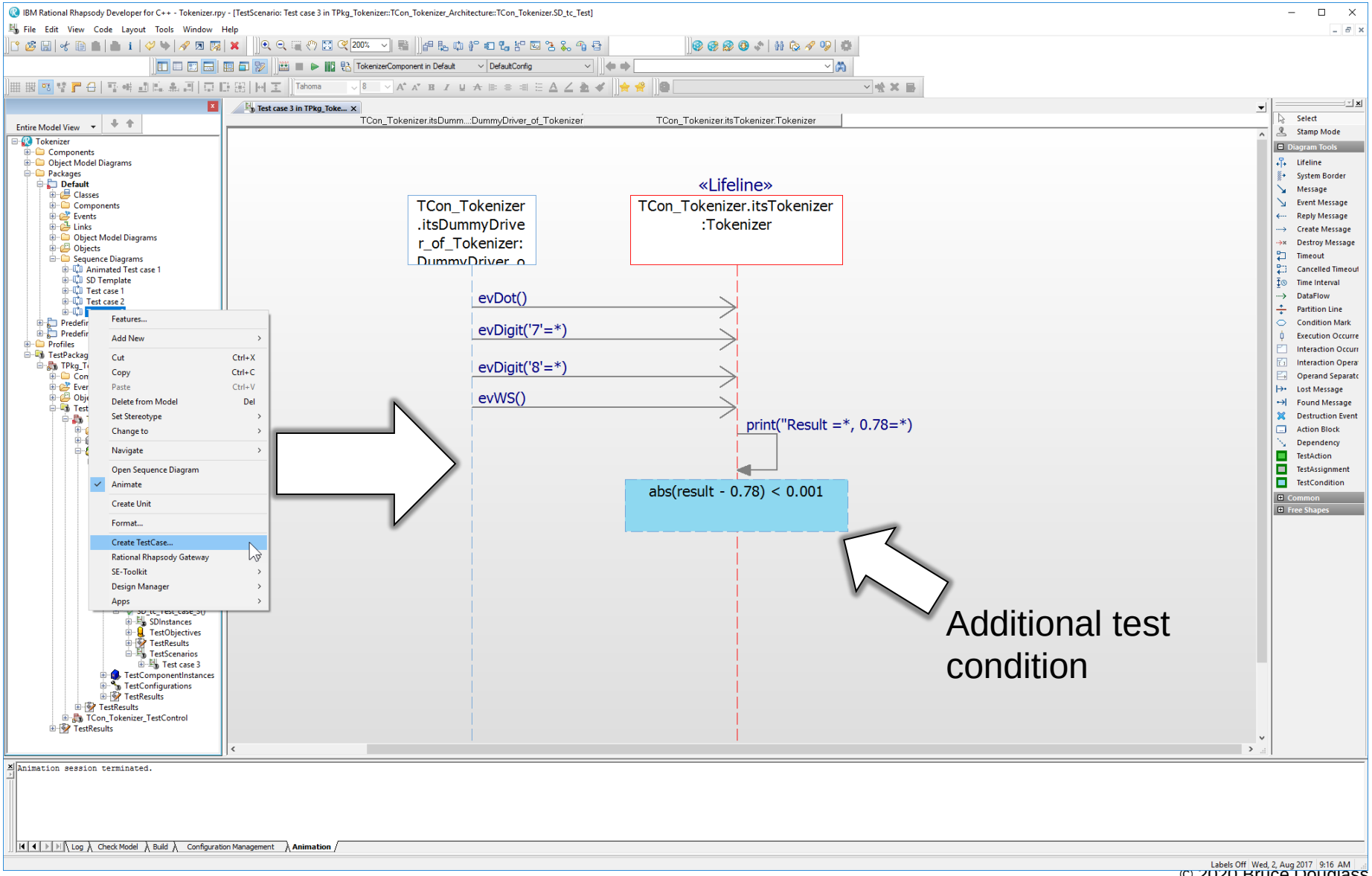
Example Model: Tokenizer (Test Conductor)



Generates

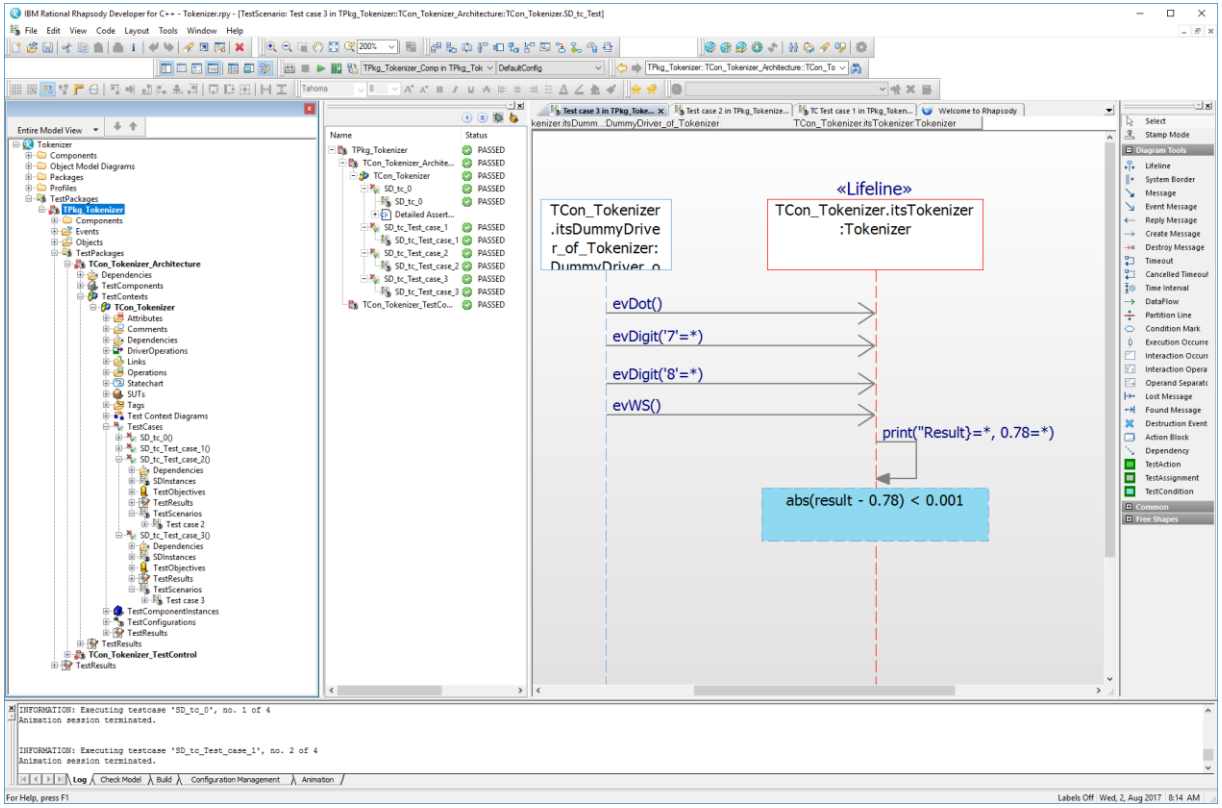


Example Model: Tokenizer (Test Conductor)



Example Model: Tokenizer (Test Conductor)

Test outcomes



Test Report

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

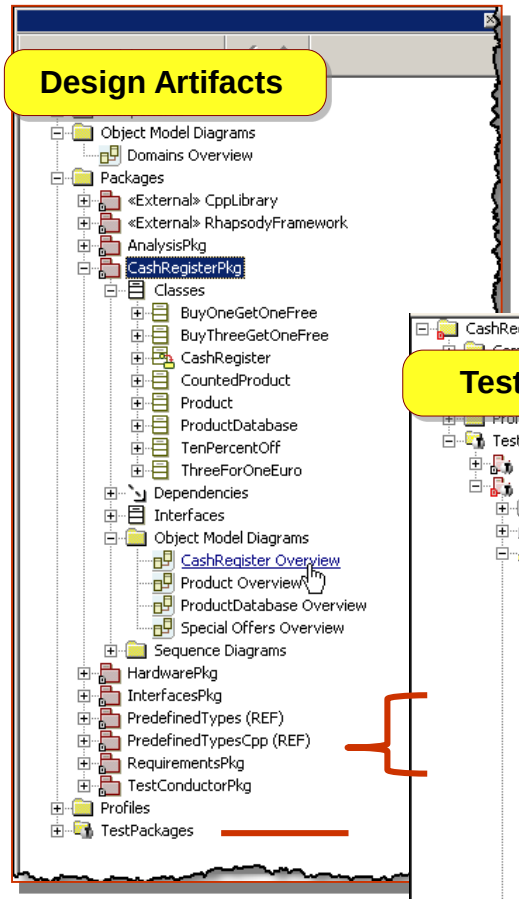
TestCase: SD_tc_Test_case_1

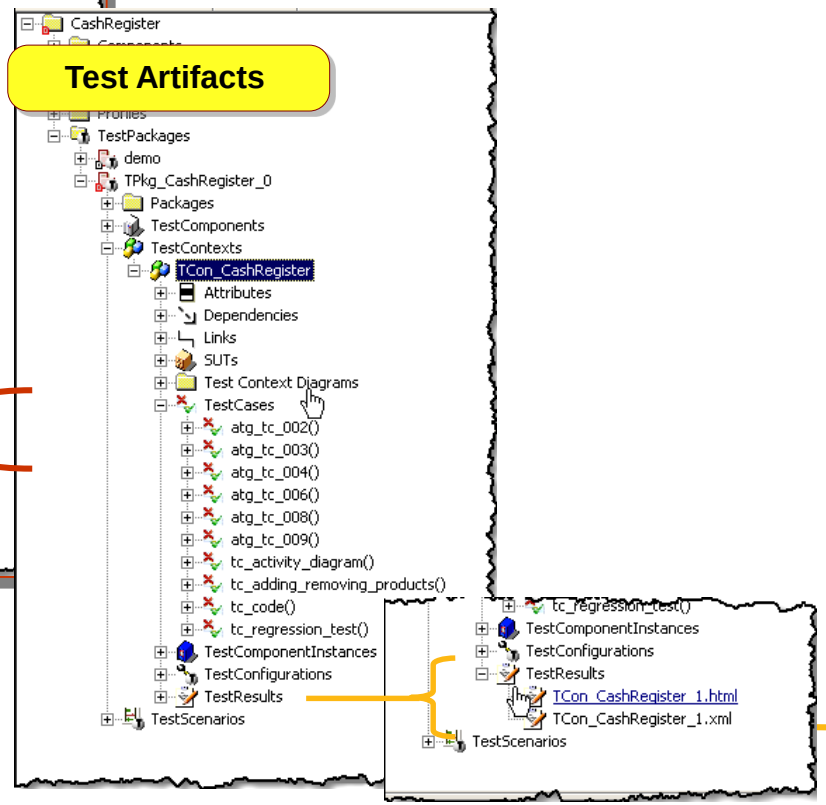
SequenceDiagram used in TestCase
TPkg_Tokenizer::TCon_Tokenizer_Architecture::TCon_Tokenizer.SD_tc_Test_case_1::Test

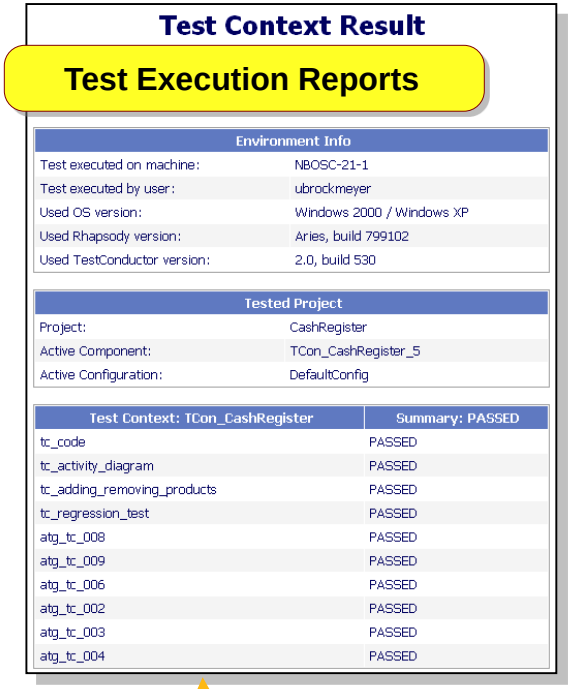
Integrated design and test environment with automation

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

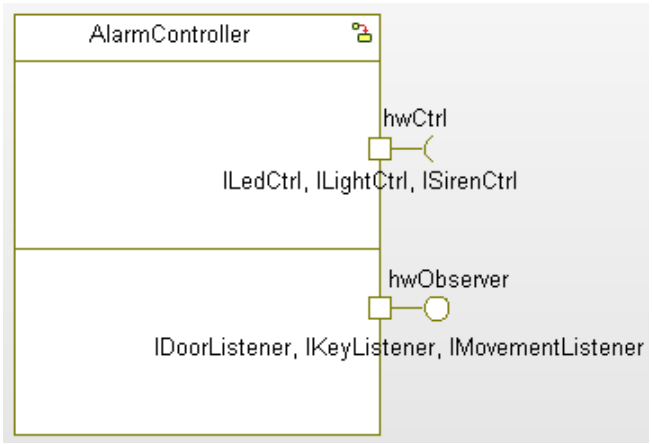


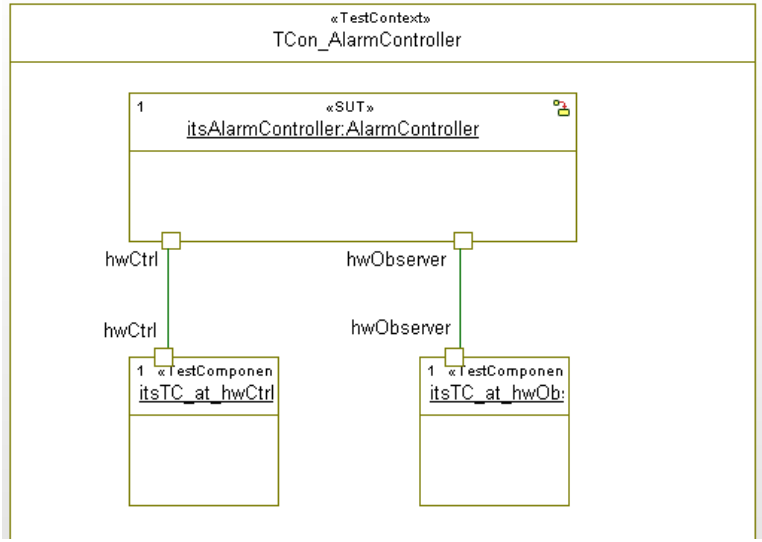




Automate quality

- 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

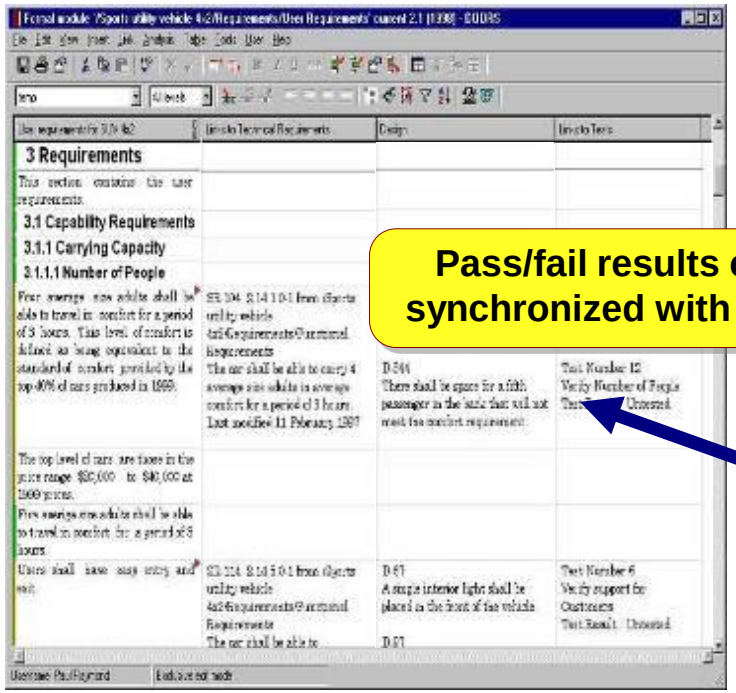




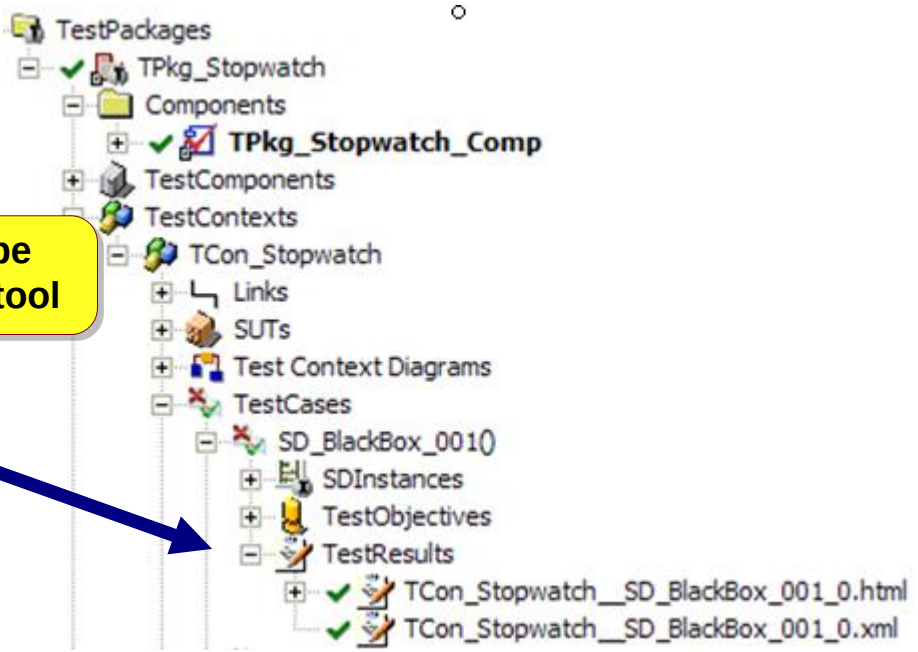
Automatically Created Test Architecture

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

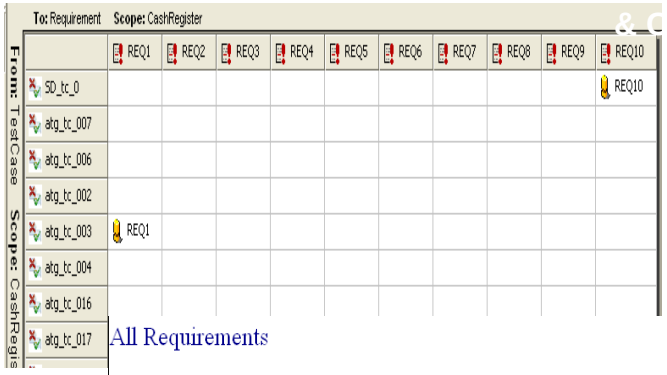


Pass/fail results can be synchronized with RM tool

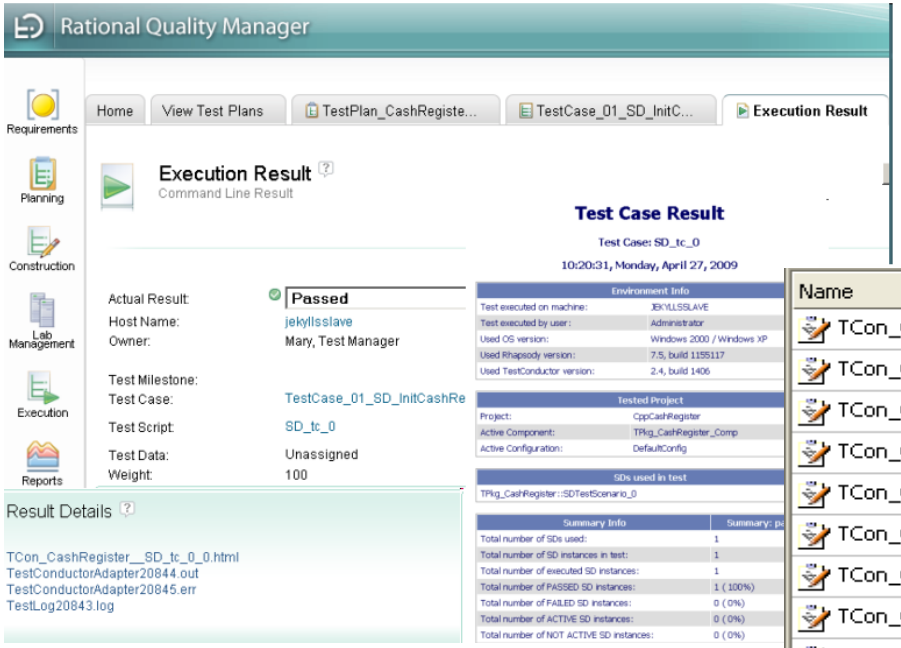


Requirements to test results coverage

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



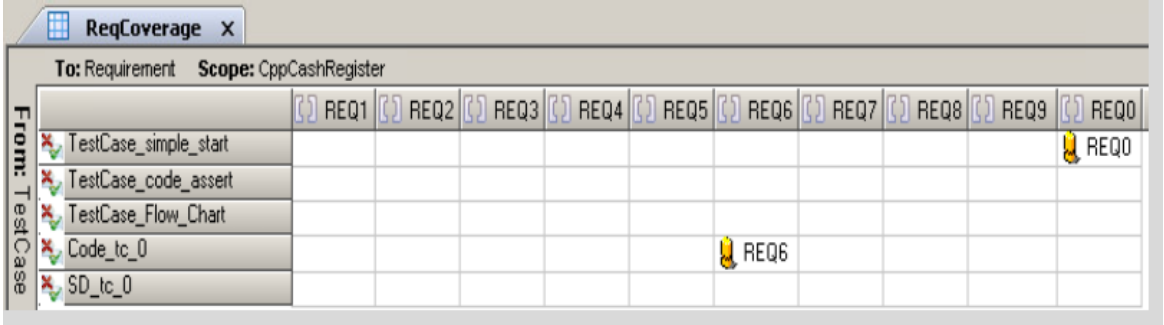
Name	Specification	Covered by Test Case
REQ1	A small stand-alone Cash Register needs to be designed that reads barcodes of products that a Customer has selected.	atg_tc_003 (Passed)
REQ10	After receiving a start event Cash Register will send a message "show(Ready)" to its display.	SD_tc_0 (Failed)
REQ2	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



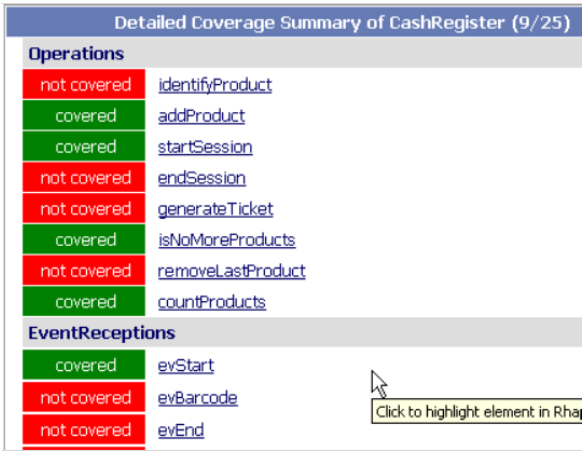
Name	Verdict
TCon_CashRegister__SD_tc_0_4.html	Failed
TCon_CashRegister__atg_tc_007_7.html	Passed
TCon_CashRegister__atg_tc_006_9.html	Passed
TCon_CashRegister__atg_tc_002_9.html	Passed
TCon_CashRegister__atg_tc_003_9.html	Passed
TCon_CashRegister__atg_tc_004_9.html	Passed
TCon_CashRegister__FC_tc_0_0.html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



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	
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%

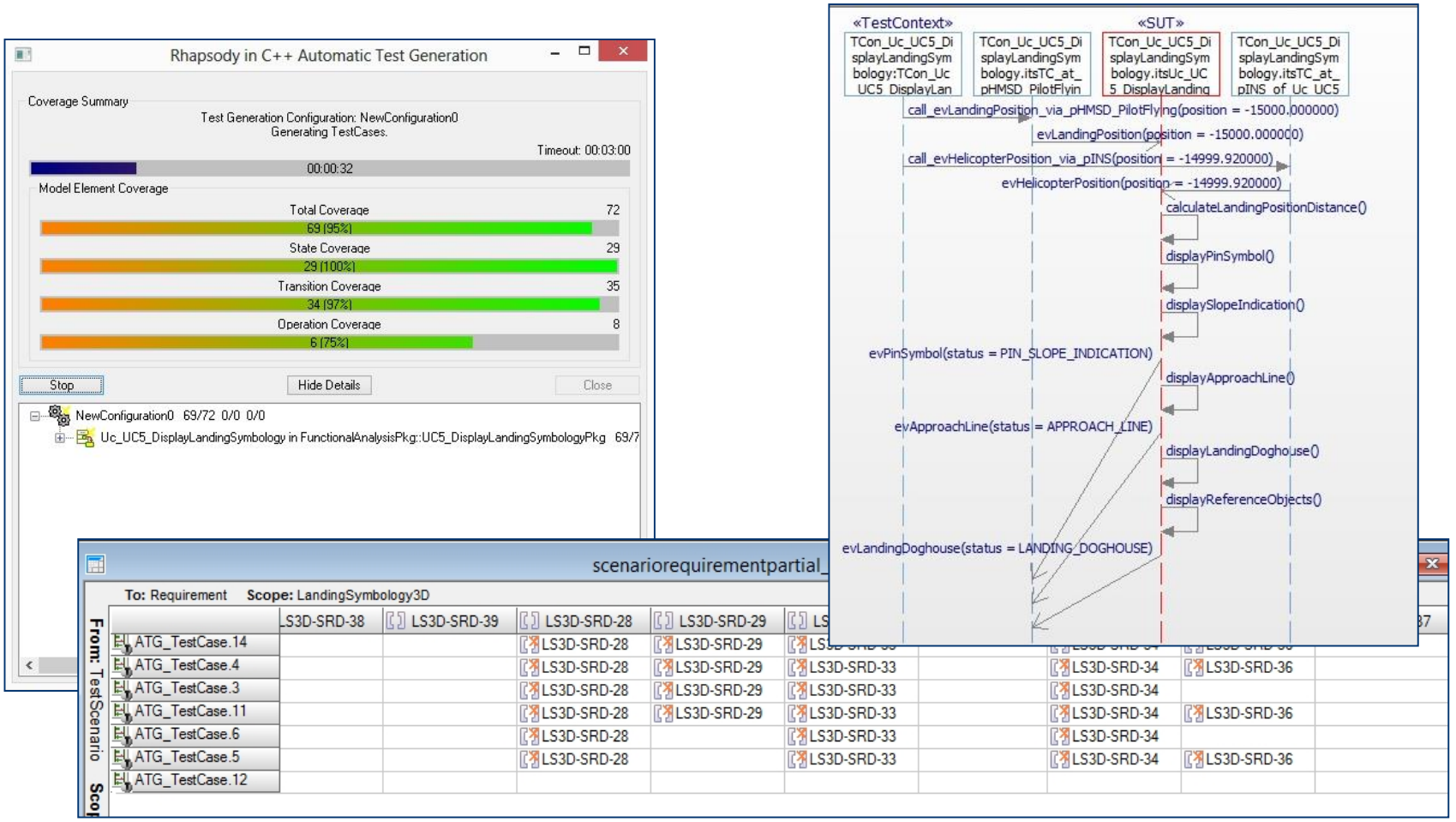
Coverage Report

Environment Info Table Of Contents Global Statistics Source Code

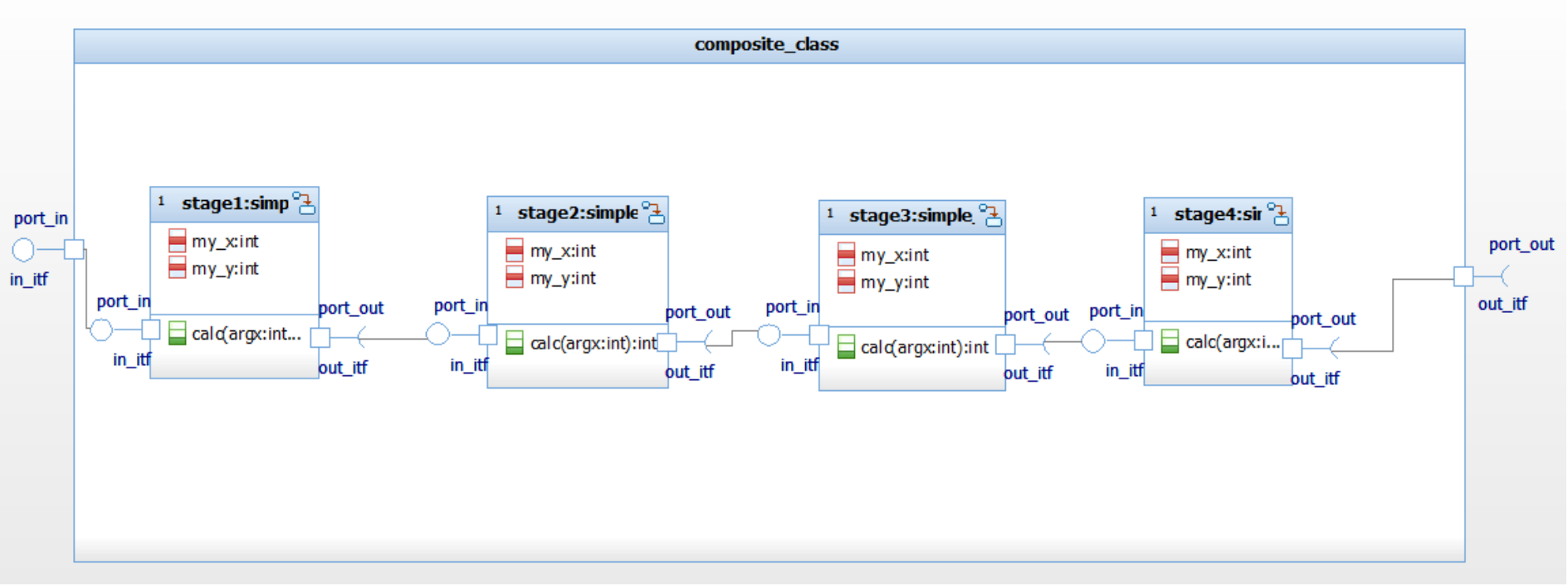
```
0          33b                                                {
           34      cleanUpRelations();
           35  }
           36
     ->    37  bool CashRegister::hw_C::InBound_C::send(IOxfEvent* event, const IOxfEventGenerationParams& params)
1          37b                                                                                                  {
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;
```

MBT – Automatic Test Generation (ATG)

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

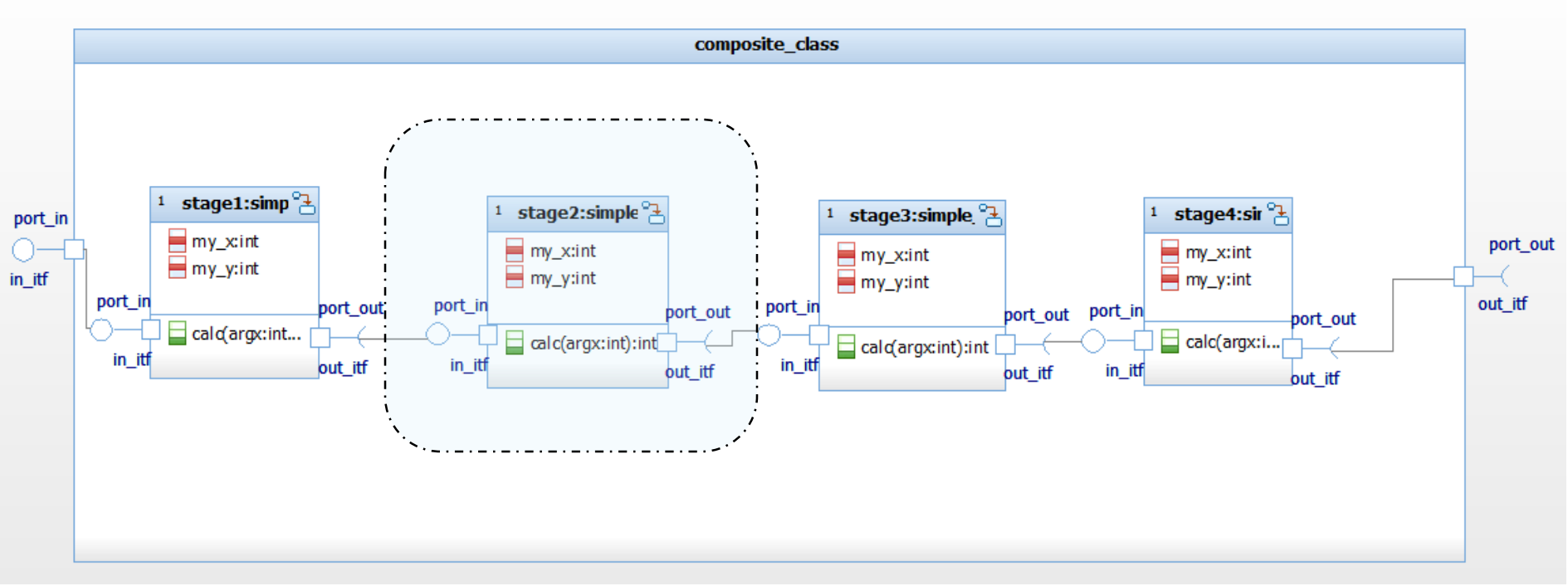


Sample System to demo MBT



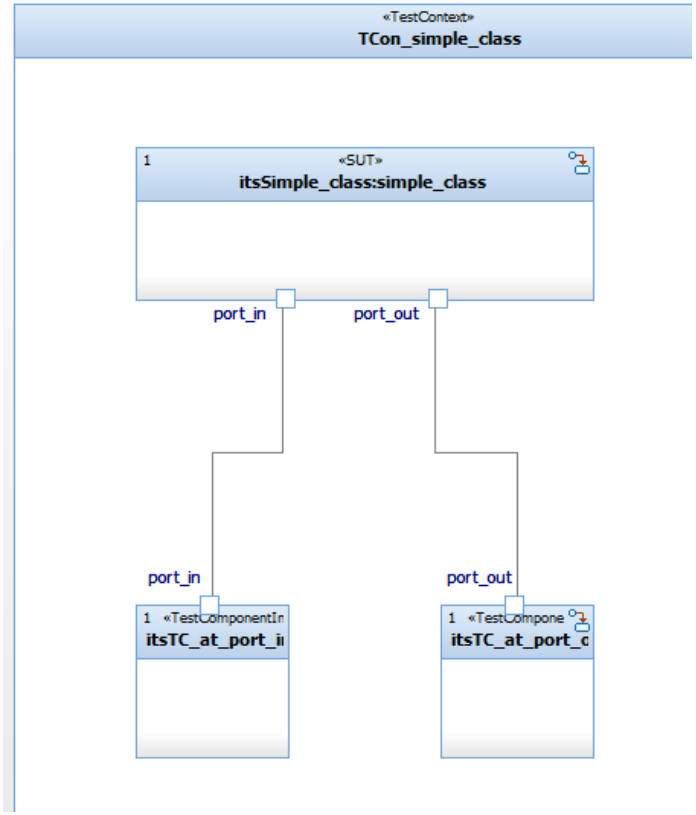
- System shows an explicitly modeled input and output interface using ports
- System contains four units with explicitly modeled input and output interfaces using ports; the units get input integer values and multiply with 2
- Software architecture shows how the units are integrated using ports and links

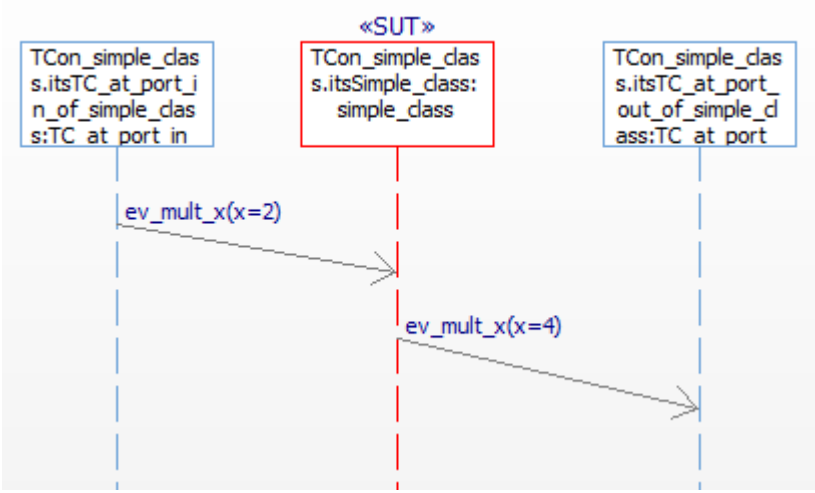
MBT: Unit Testing I



- 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

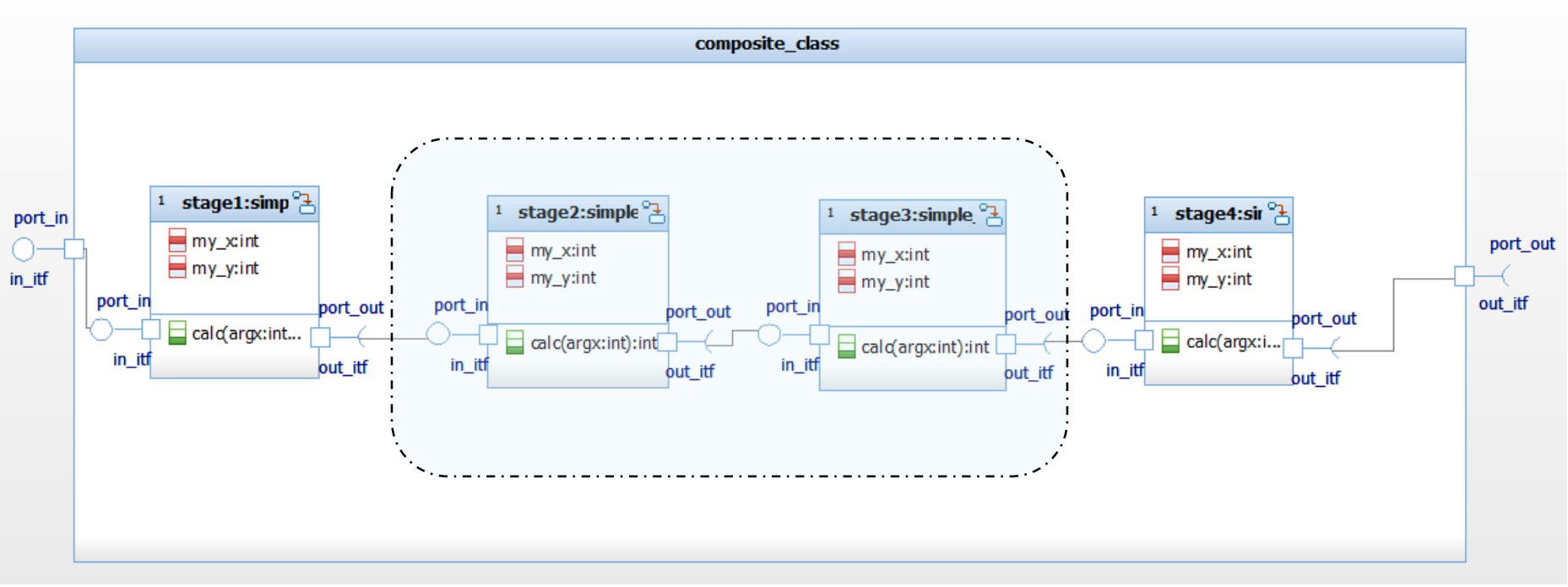
MBT: Unit Testing II





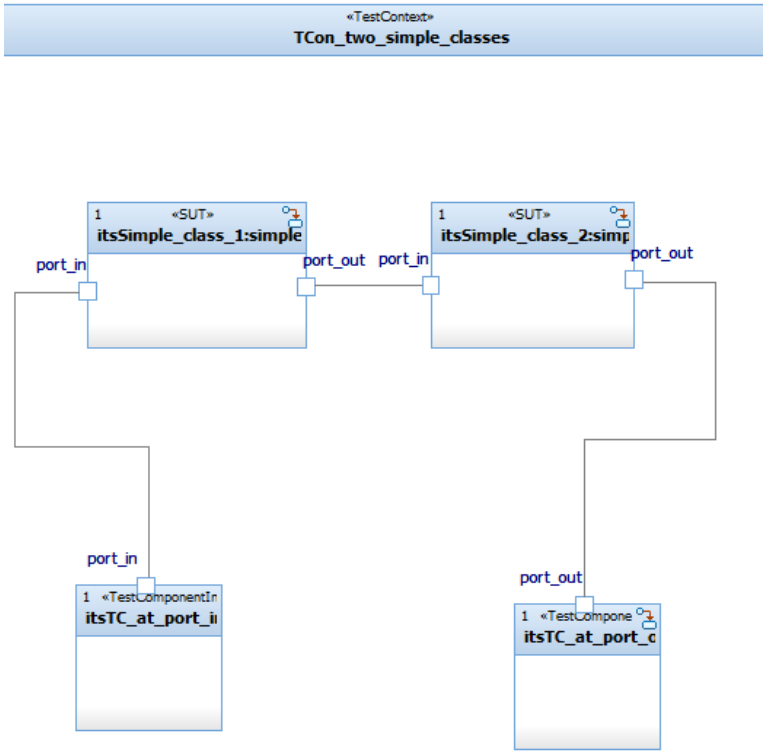
- An instance of the unit under test (SUT) is contained in the test architecture, and two test components which are connected to the ports of the SUT
- Developers specify the expected input / output behaviour in a test case
- TestConductor executes the unit tests and computes test verdicts (pass/fail)

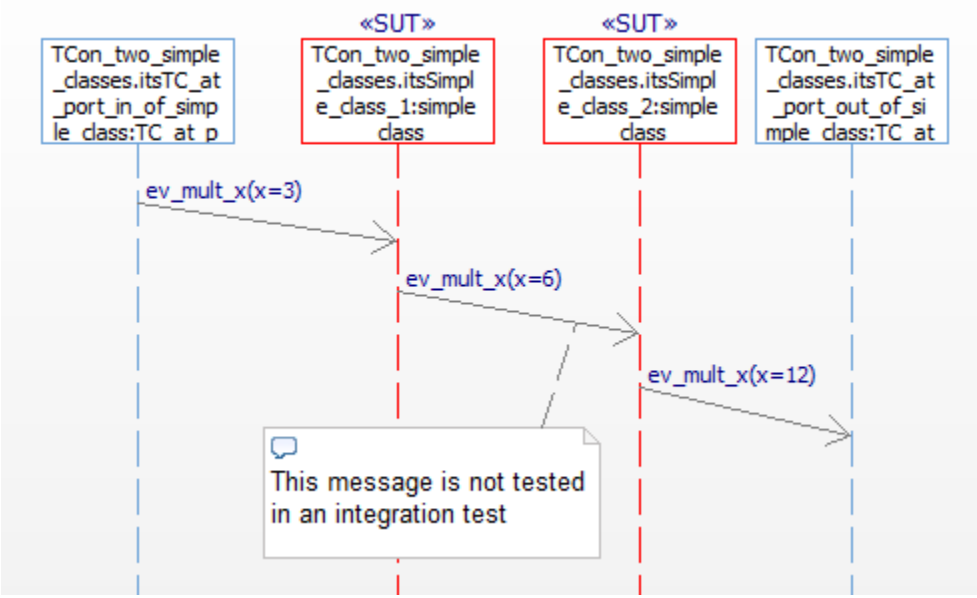
MBT: IntegrationTesting I



- 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

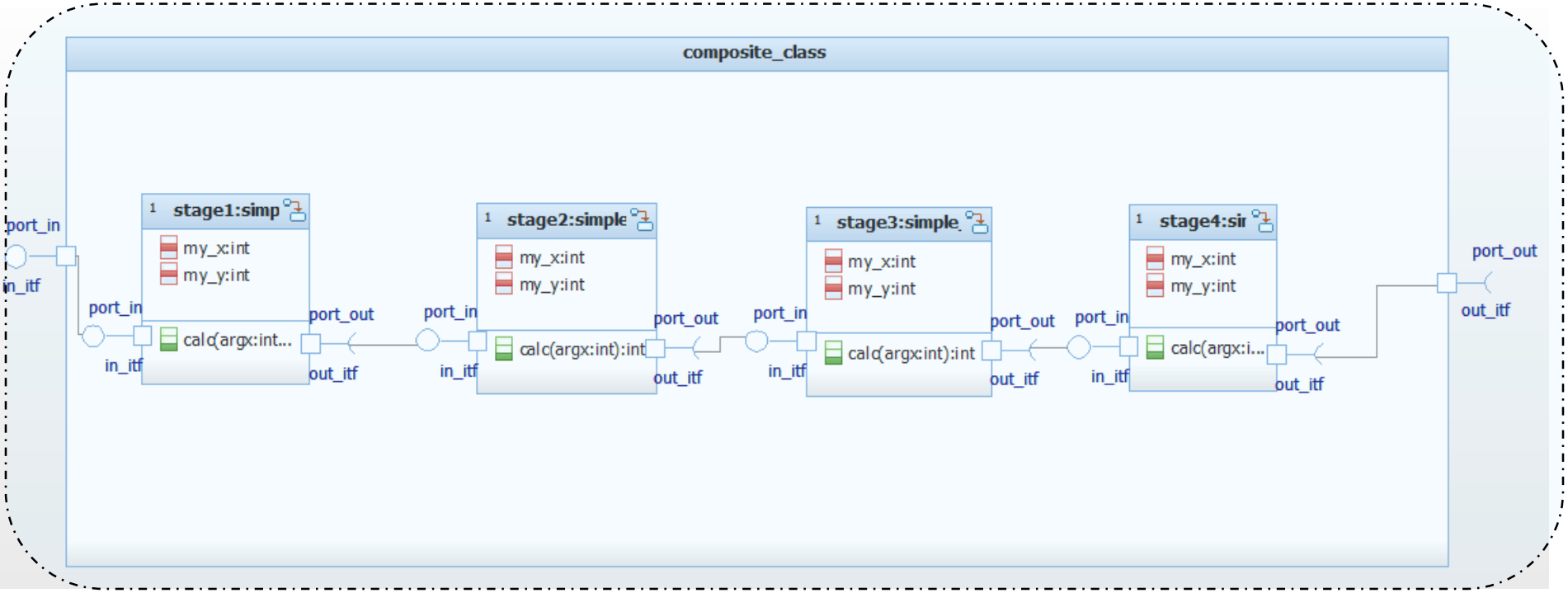
MBT: Integration Testing II





- 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)

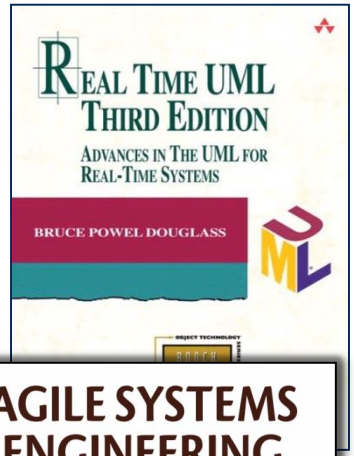
MBT: Software System Testing I

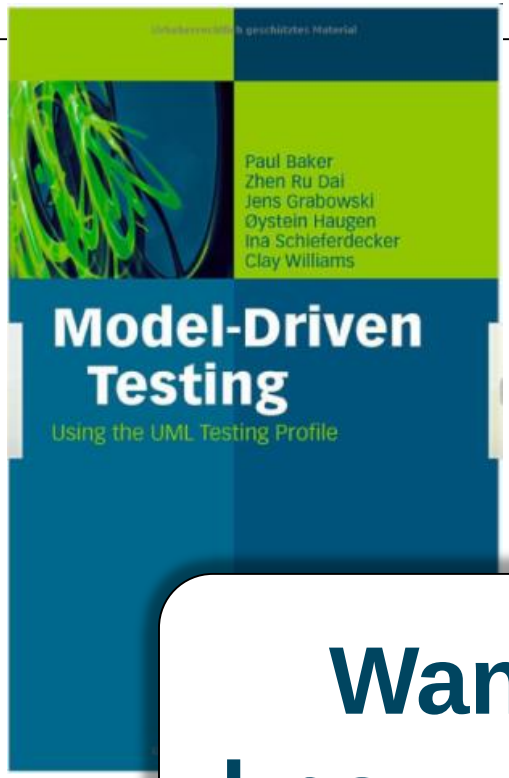


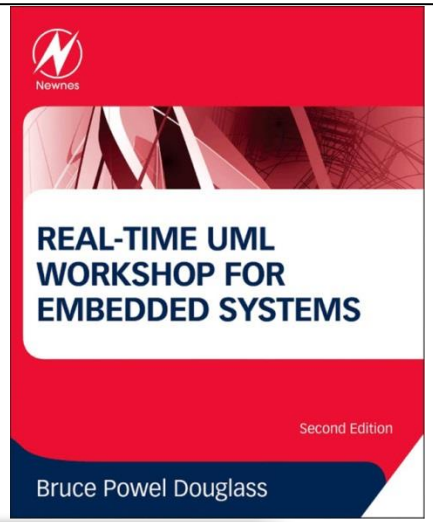
- Objective is to test the whole SW system on host *or on an embedded target*
- TestCondcutor automatically creates test architectures for the SW system using the system ports and interfaces
- “Black box test”
  - requirements based testing using the interfaces of the SUT

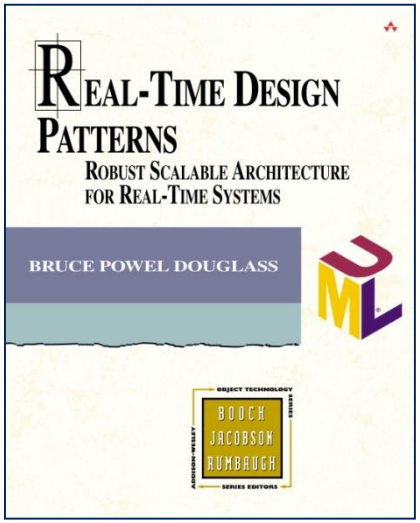
Summary

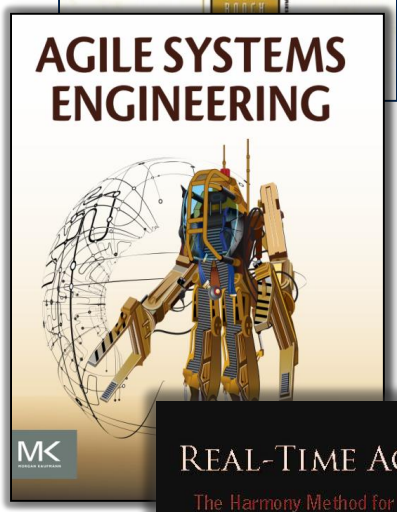
- Testing is hard!
- Models are simplifications of reality that allow us to focus on relevant issues
- Models provide significant enhancement to our ability to deal with engineering data, such as requirements, design, and implementation
- Models likewise enhance our ability to test:
  - Development of 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

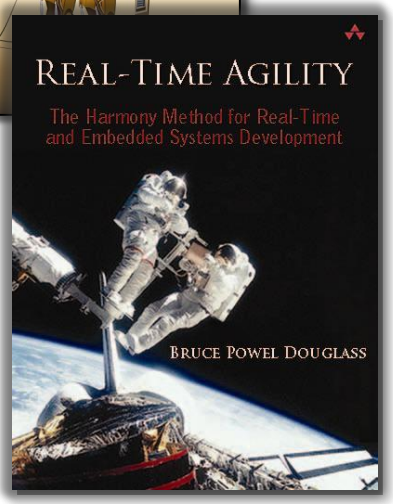












Want to know more?