

Rational. Rhapsody

## Rhapsody in C++ Tutorial

for

IBM<sup>®</sup> Rational<sup>®</sup> Rhapsody<sup>®</sup>  
TestConductor Add On

Rational. Rhapsody

IBM.



## License Agreement

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of the copyright owner, BTC Embedded Systems AG.

The information in this publication is subject to change without notice, and BTC Embedded Systems AG assumes no responsibility for any errors which may appear herein. No warranties, either expressed or implied, are made regarding Rhapsody software including documentation and its fitness for any particular purpose.

## Trademarks

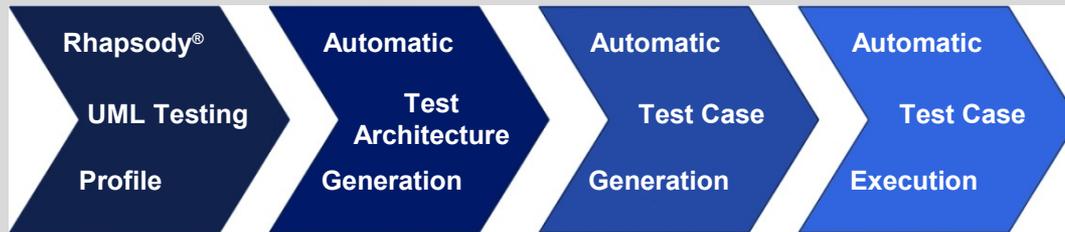
IBM® Rational® Rhapsody®, IBM® Rational® Rhapsody® Automatic Test Generation Add On, and IBM® Rational® Rhapsody® TestConductor Add On are registered trademarks of IBM Corporation.

All other product or company names mentioned herein may be trademarks or registered trademarks of their respective owners.

© Copyright 2000-2016 BTC Embedded Systems AG.  
All rights reserved.

# TestConductor for Rhapsody in C++

**In this tutorial** we would like to give you an impression of the Rhapsody Testing Environment, which goes beyond current embedded software testing technologies; it ensures that the system can be continuously tested throughout the design process. The Testing Environment and its parts seamlessly integrate in Rhapsody UML and guide the user through the complex process of test preparation, execution and result analysis.

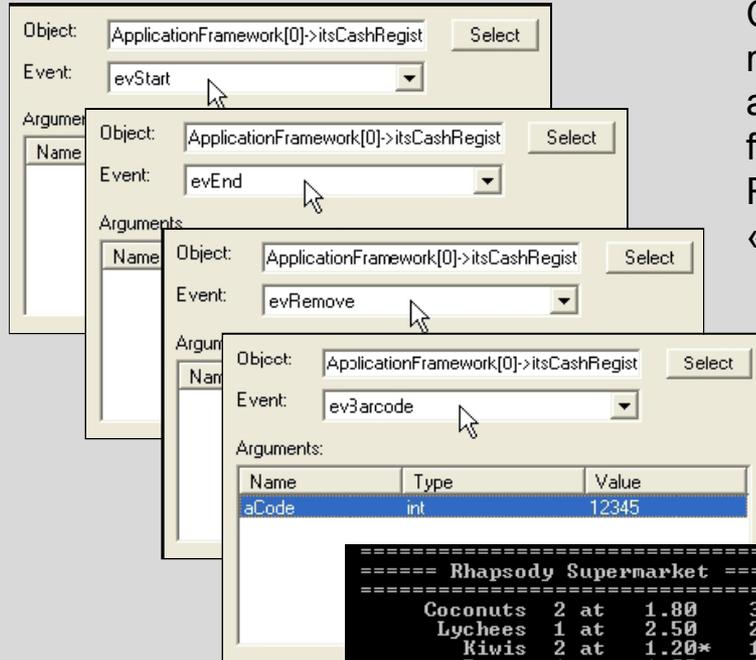


*IBM® Rational® Rhapsody® Testing Environment*

**TestConductor** is the test execution and verification engine in the Rhapsody Testing Environment. It executes test cases defined by sequence diagrams, flow charts, statecharts, and source code. During execution TestConductor verifies the results against the defined requirements.

**Rhapsody ATG** is the **A**utomatic **T**est **G**eneration engine in the Rhapsody Testing Environment. In order to thoroughly verify the functionality of the System Under Test (SUT), it uses the UML model information as well as the generated source code as basis for analysis, and creates executable test cases with high coverage rates. See separate ATG tutorial how to use ATG.

# CashRegister Application



```
===== Rhapsody Supermarket =====
=====
Coconuts 2 at 1.80 3.60
Lychees 1 at 2.50 2.50
Kiwis 2 at 1.20* 1.20
Pears 1 at 1.00 1.00
Pomegranates 1 at 1.99 1.99
Watermelons 2 at 3.50 7.00
=====
Total = 17.29
=====
```



The **CashRegister application**, the example C++ application for this tutorial, models a simple cash register. Make yourself familiar with the use cases of the application. Open the project „CppCashRegister“ from the folder „Samples/CppSamples/TestConductor“ in your Rhapsody installation, run the component «CashRegisterNoGui», and use the following input:

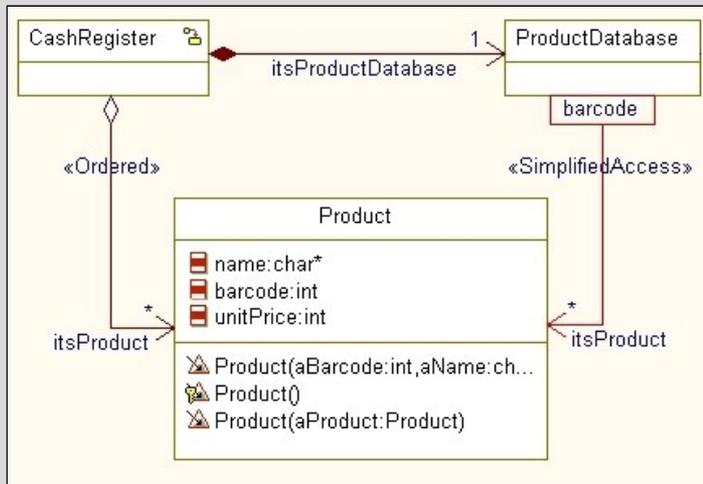
**To create** a new shopping basket send the event evStart to ApplicationFramework[0]->itsCashRegister.

**To add** an product to the shopping basket send the event evBarcode to ApplicationFramework[0]->itsCashRegister. The event evBarcode needs the product code as argument. The product database knows codes between 12344 and 12349.

**To remove** the last added product from the shopping basket send the event evRemove to ApplicationFramework[0]->itsCashRegister.

**To print** the bill send the event evEnd to ApplicationFramework[0]->itsCashRegister.

# CashRegister Model

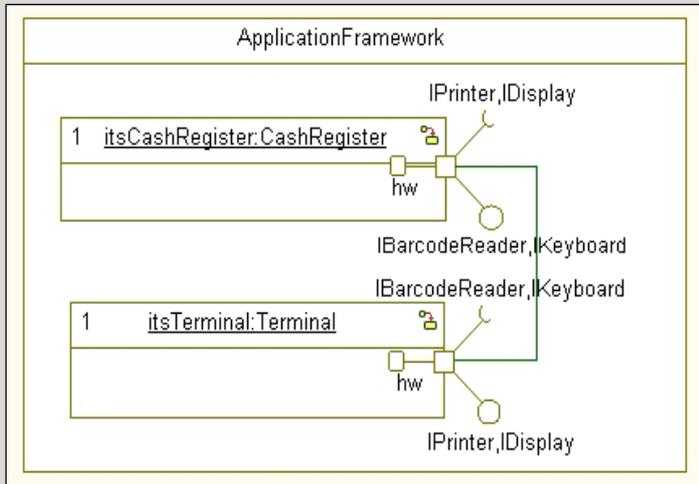


The **CashRegister model** mainly contains the CashRegister class, a list of selected products, and a product database class with a list of all products ordered by barcode numbers. The model delegates all input and output messages to classes with interfaces of IDisplay, IPrinter, IBarcode and IKeyboard. These classes are connected by a port named „hw“ to the CashRegister-class.

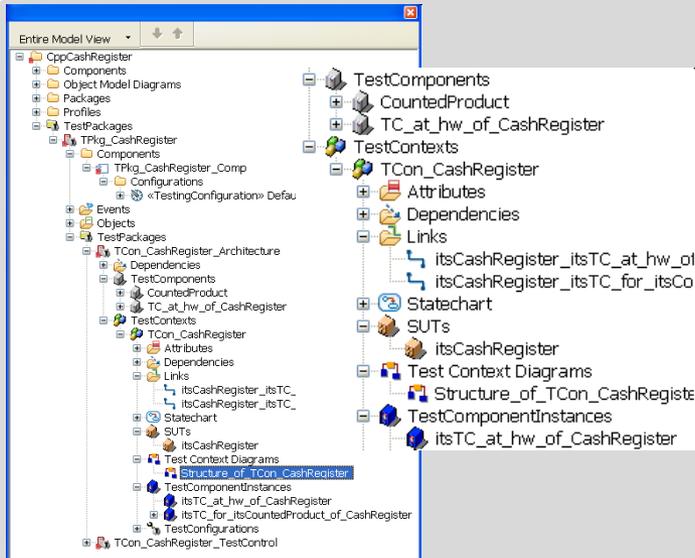
The **ApplicationFramework class** initialises its parts itsCashRegister of type CashRegister and itsTerminal of type Terminal. The link between the parts ensures the bi-directional communication over the port hw.

The **CashRegister class** is able to manage the list of products the user wants to buy. View the provided state chart to get familiar with event processing and state changes.

The **Terminal class** provides the interfaces IPrinter and IDisplay. Imagine the Terminal class as an input/output terminal, which is able to process keyboard inputs and displays the progress and the bill.

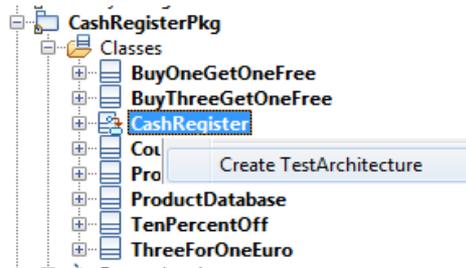


# System Under Test



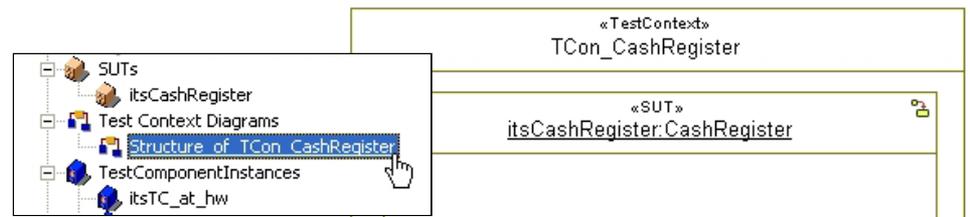
Defining the System Under Test (SUT) is the first step in the test workflow. This tutorial will focus on the CashRegister class. To define CashRegister to be the SUT, we have to create a test architecture. The needed administrative framework will be placed in the folder „TestPackages“.

The System Under Test (SUT) is a part and is the component being tested. A SUT can consist of several objects. The SUT is exercised via its public interface operations and events by the test components.



1

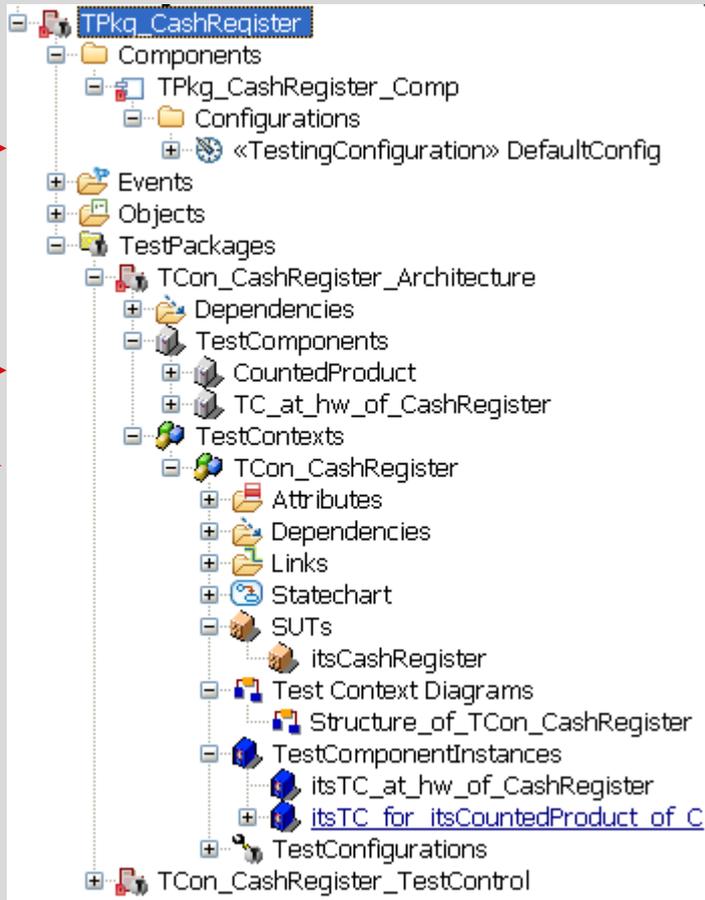
Select the class „CashRegister“ in the browser and choose from context menu „Create TestArchitecture“.



2

Have a look on the newly created Test Context Diagram „Structure\_of\_TCon\_CashRegister“, and view the resulting parts in the composite class „TCon\_CashRegister“ of our test context.

# Test Architecture

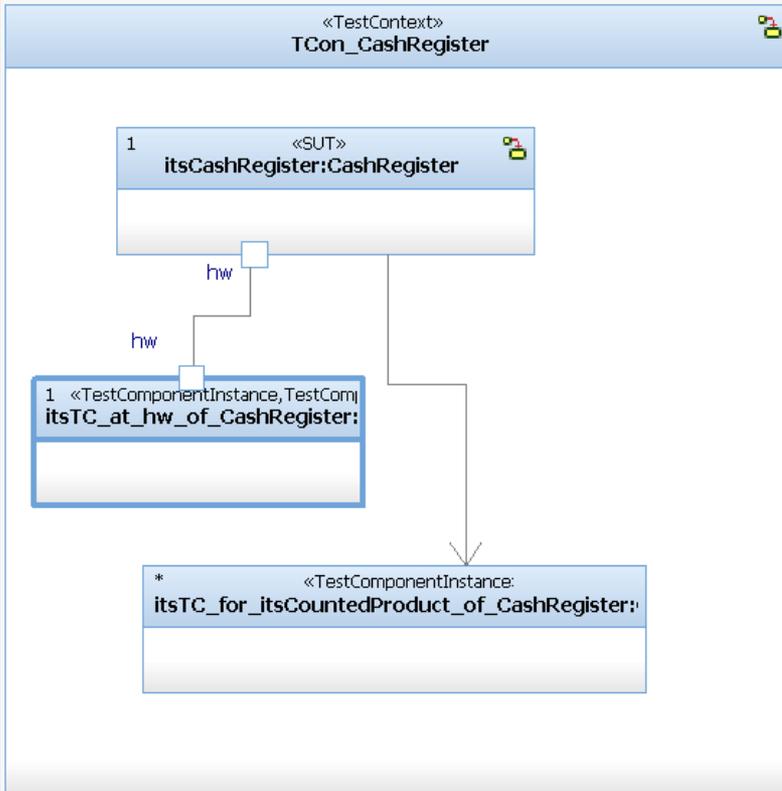


The automatically created test architecture is completely represented in the browser and seamlessly integrates into Rhapsody; think of it as an independent test model besides the design model. After creation the following elements are visible:

The new configuration under the component „TPkg\_CashRegister\_Comp“ initializes the test components and SUT objects and their interconnections when a test case is started.

A test component is a class of a test system. Test component objects (test component instances) realize partially the behavior of a test case. A test component might have a set of interfaces via which it might communicate via connections with other test components or with SUT objects.

A test context describes the context in which test cases are executed. It is responsible for defining the structure of the test system. The test components and SUT objects are normally parts of a test context.



The automatically created test context represents the formal structure of the test system. TestConductor analyzed the model structure in consideration of the selected SUT and proposed a test structure, which is visualized in the test context diagram inside the test context. TestConductor generated corresponding test components for ports and associations of the SUT.

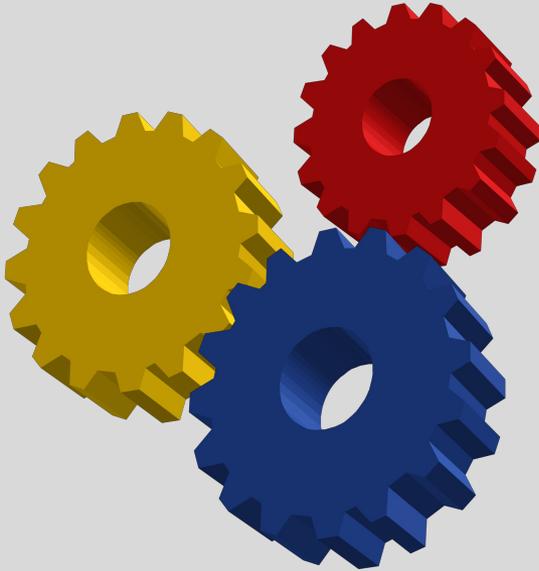
## The composite class

„TCon\_CashRegister“ is the part container for the SUT object and the created test component objects.

The class „TC\_at\_hw“ realizes the required interfaces „IDisplay“ and „IPrinter“ of port „hw“. Using ports as a high-grade encapsulation mechanism will result in clean test architectures.

The class „CountedProduct“ is a derivation of the design class „CountedProduct“. It is generated due to its association to the CashRegister class.

Test cases are the soul of a test system. Until now we created a complete test architecture around the SUT with a few mouse clicks in less than a minute. The established and reviewed test system is linkable and runnable. Well, the body works, let's have a look at the test cases. A test case ...



**is a specification** of one case to test the system including what to test, with which inputs, and what the expected outcomes are. It is defined in terms of stimuli injected to SUT objects and observations coming from SUT objects.

**is an operation** of a test context that specifies how a set of cooperating test components interact with the SUT.

**can be specified** as sequence diagrams, flow charts, statecharts, and source code.

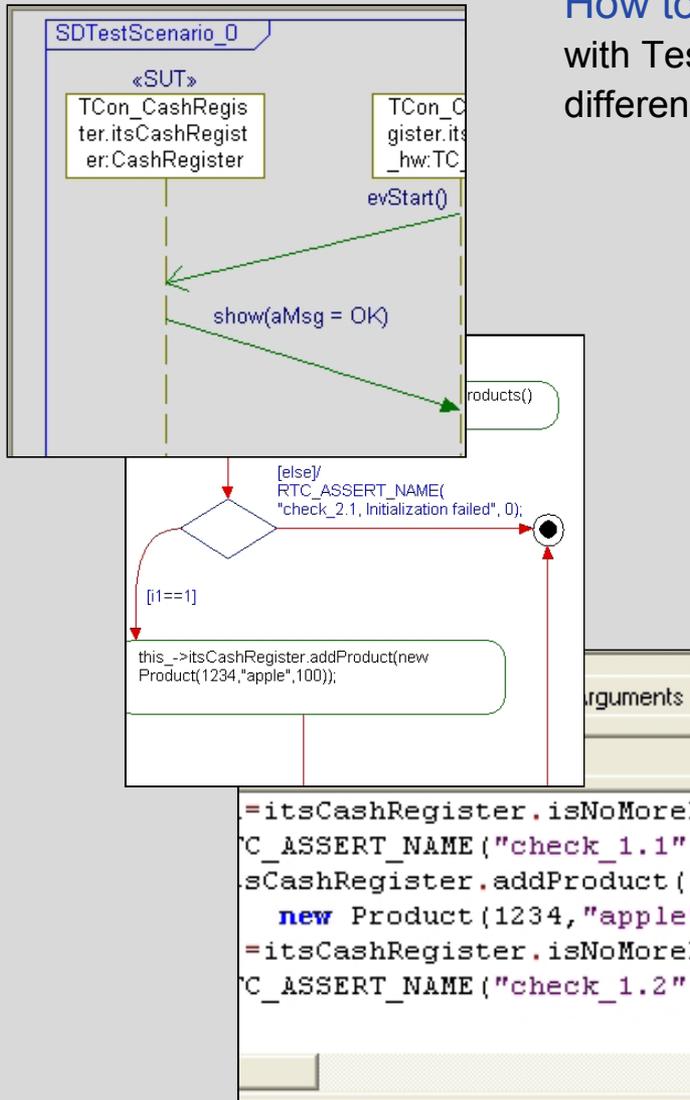
**can be generated** automatically by using TestConductor's test case wizard.

**can be generated** automatically with the Rhapsody Automatic Test Generation (ATG).

**can be recorded** as animated sequence diagrams.

**can be created** by hand.

# Test Case Specification



How to manually create test cases and how to execute them with TestConductor will be discussed in the following sections. The different kinds of definitions have their own strengths:

**Sequence diagram** test cases can be recorded automatically or created by hand. In some cases they have already been specified during the analysis phase of the project, and define the actions and reactions of the SUT. The graphical formalism boosts the readability and understanding.

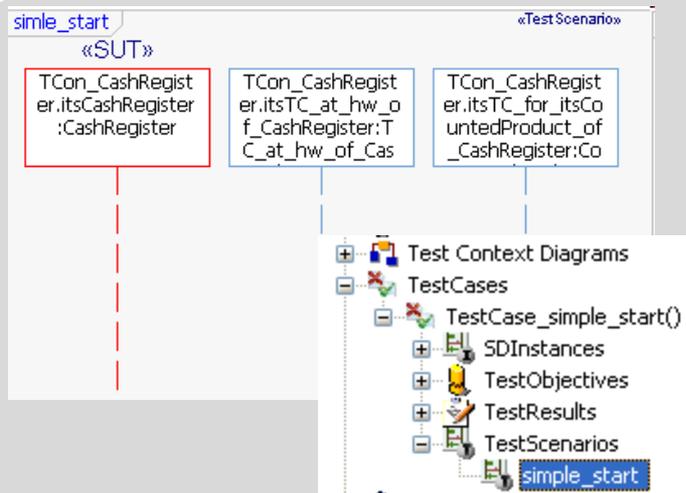
**Flow chart** test cases also benefit from their graphical nature, but in contrast to sequence diagrams the use of complex data types (structs) and control structures (if-then-else) is supported out-of-the-box.

**Statechart** test cases are a well known and convenient means to specify behavior based on states and modes.

**Source code** test cases are often preferred by experienced programmers.

**In summary** TestConductor, the Rhapsody test case execution engine, works with all kinds and combinations of test case definitions.

# Test Case: Sequence Diagram I

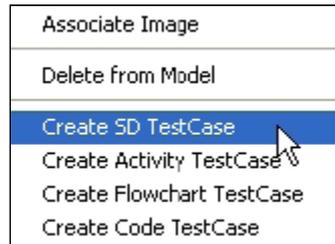


To manually create a sequence diagram test case we have to define a test scenario which is represented as a sequence diagram and link it to a test case. TestConductor simplifies this process with a single command.



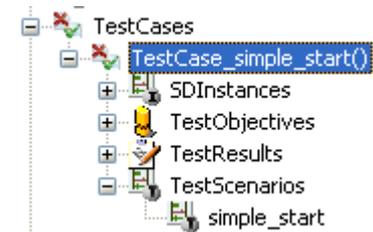
1

Select the test context „TCon\_CashRegister“ in the Rhapsody-Browser ...



2

... and choose from the context menu „Create SD TestCase“..

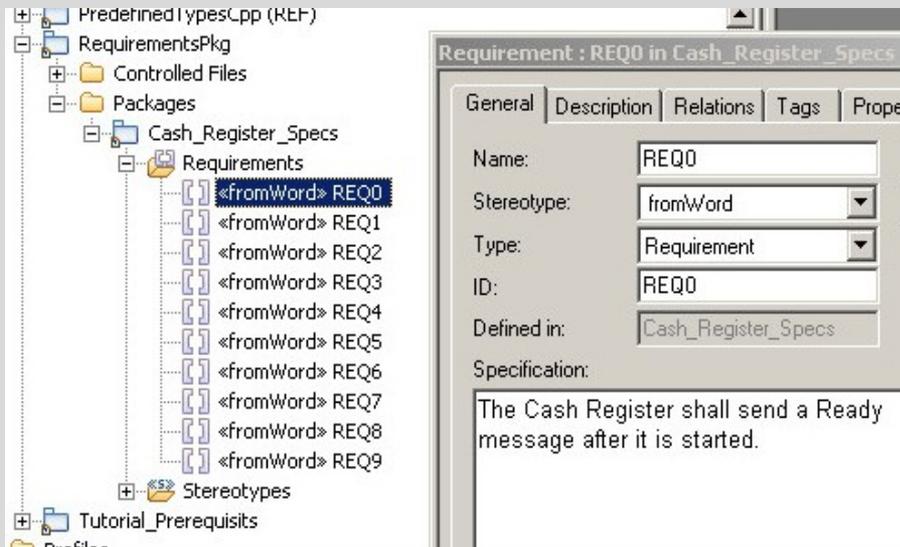


3

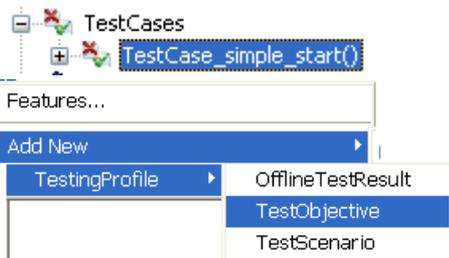
Rename the test case to „TestCase\_simple\_start“. Rename the test scenario to „simple\_start“ and open it.

# Test Case: Sequence Diagram II

1  
2

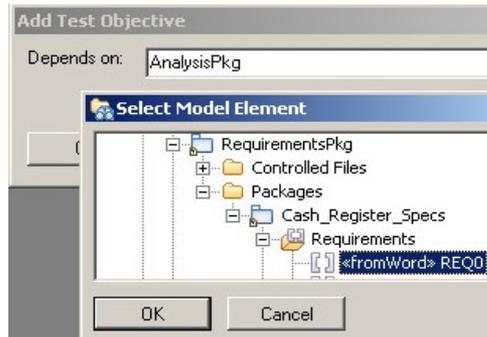


Determine the test objective of the test case: the SD test case should check that requirement “REQ0” is indeed fulfilled by the CashRegister class. To make explicit that the SD test case shall verify this particular requirement, a test objective is added to the test case.



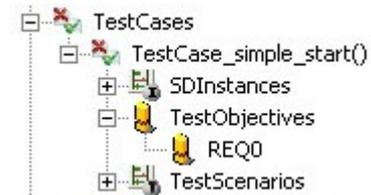
1

Select the test case and select “Add New -> TestingProfile -> TestObjective”



2

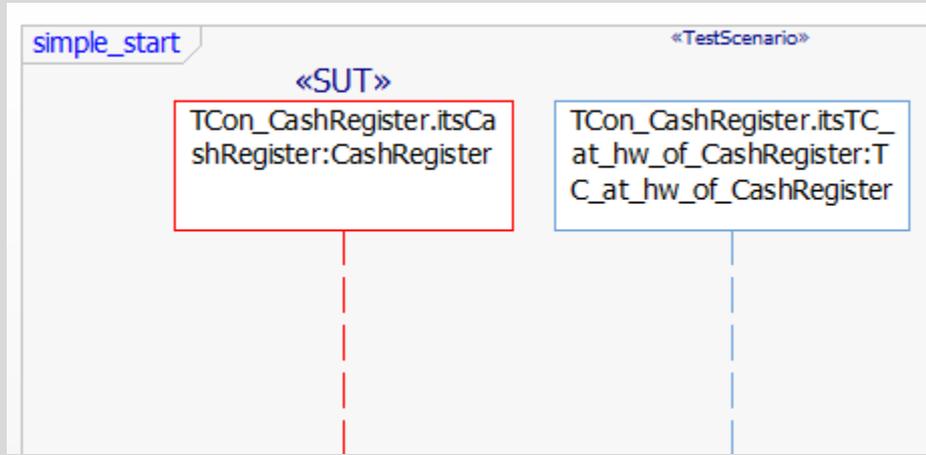
Select requirement “REQ0” as target of the test objective”



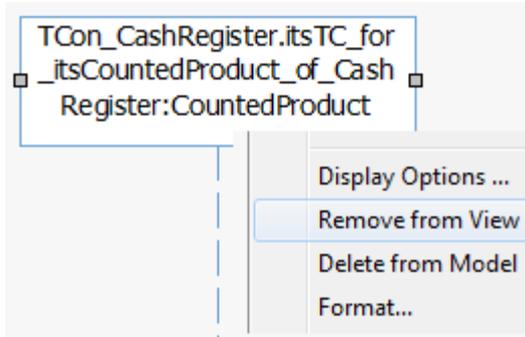
3

The test objective now links the test case to the requirement “REQ0”.

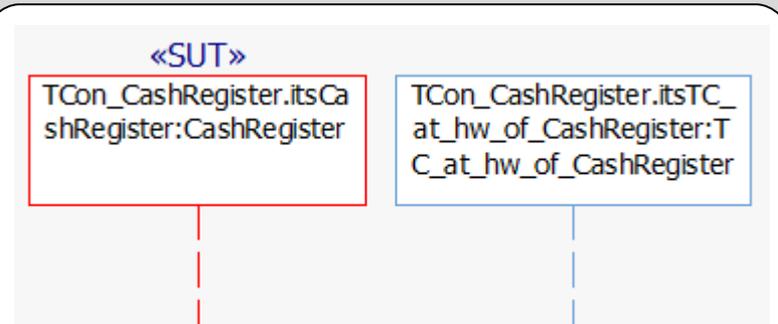
# Test Case: Sequence Diagram III



Determine the involved objects for the desired test scenario and remove not needed instance lines from the view in order to establish action and reaction between remaining instances.

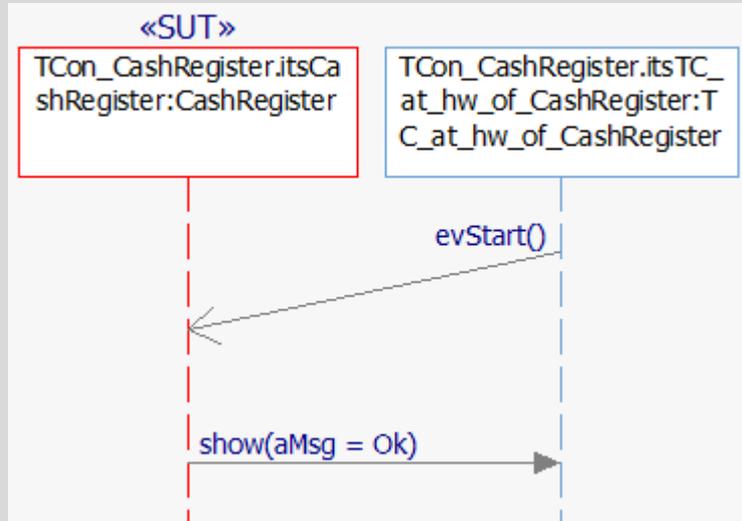


**1** Right click the instance line „itsTC\_for\_itsCountedProduct“ and remove it from view.



**2** Arrange the remaining instance lines „itsTC\_at\_hw\_of\_CashRegister“ and „itsCashRegister“.

# Test Case: Sequence Diagram IV



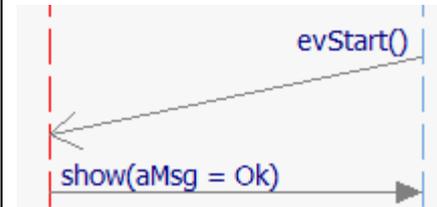
Define **action and reaction** of the system under test. We will specify the „simple\_start“ scenario, where the user sends the event `evStart()` to the SUT, and the SUT shall react with a status message `show(aMsg)`. TestConductor, the execution engine, shall act as **as driver** for `evStart()`, and **as observer** for `show(aMsg)`. **Driving means** to simulate e.g. the users activity during test execution by automatically sending the message to the SUT in order to provoke a reaction. The test will pass, if TestConductor observes the specified reaction from the SUT. Otherwise it will fail.

```
CashRegister::evStart()
CashRegister::evBarcode()
identifyProduct(int)
addProduct(Product)
startSession()
CashRegister::evEnd()
endSession()
```

**1** Draw the driving message „evStart()“ from „itsTC\_at\_hw\_of\_CashRegister“ to the SUT „itsCashRegister“.

```
TC_at_hw::print(char*)
TC_at_hw::show(char*)
IPrinter::print(char*)
 IDisplay::show(char*)
```

**2** Draw the message „show()“ from the SUT „itsCashRegister“ to „itsTC\_at\_hw\_of\_CashRegister“ such that it can be observed.



**3** Specify the parameter `aMsg` by editing the label of `show()` to „show(aMsg = OK)“.

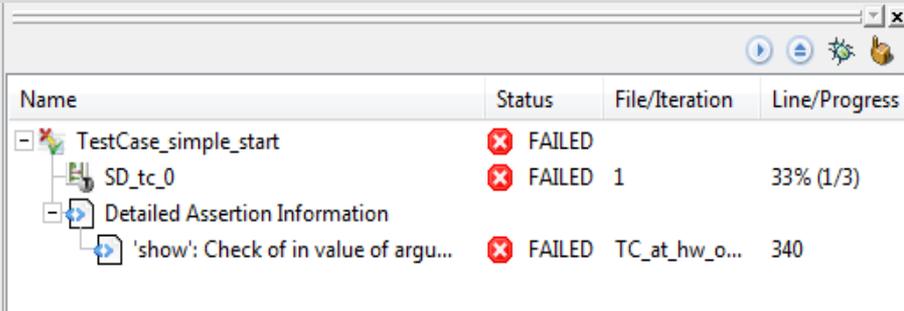
# Test Case Execution I

Execute the test case with Rhapsody TestConductor. The execute dialog lists all executed test scenarios, their progress and status.

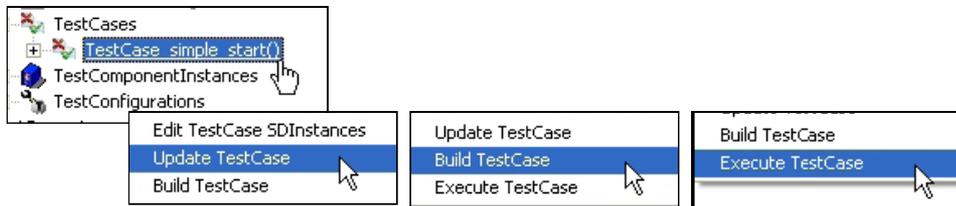
The status, the final result can be either „PASSED“ or „FAILED“.

The progress displays how many steps are finished yet. In case of a passed test 100% have to be achieved.

The buttons in the top right corner of the execution dialog can be used to control actual test case execution and will be explained later.

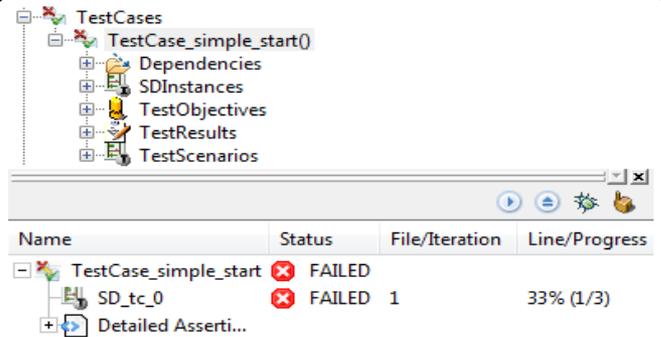


Name	Status	File/Iteration	Line/Progress
TestCase_simple_start	FAILED		
SD_tc_0	FAILED	1	33% (1/3)
Detailed Assertion Information			
'show': Check of in value of argu...	FAILED	TC_at_hw_o...	340



1

To execute the test case with TestConductor select the test case „TestCase\_simple\_start“ and choose from the context menu the items „Update TestCase“, „Build TestCase“, and „Execute TestCase“. The Rhapsody TestConductor execution dialog will open.

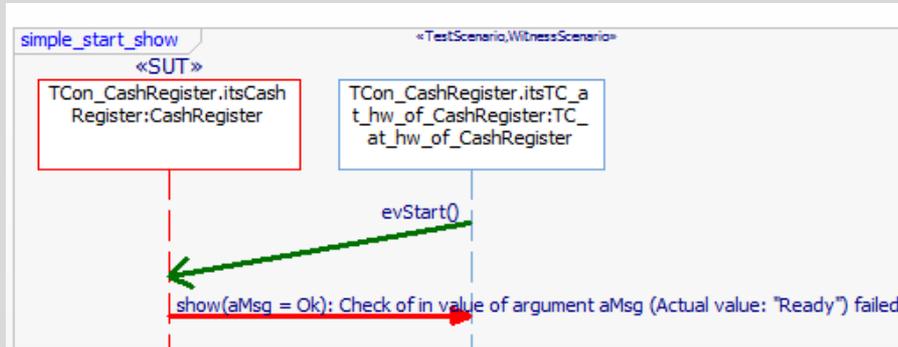


Name	Status	File/Iteration	Line/Progress
TestCase_simple_start	FAILED		
SD_tc_0	FAILED	1	33% (1/3)
Detailed Asserti...			

2

The test case execution dialog is a dockable dialog that can be placed e.g. underneath the main browser window.

# Test Case Execution II



The test case execution **FAILED** with Rhapsody TestConductor. To analyze the reason TestConductor offers two kind of views. The HTML-report displays a textual summary and can be found directly under the test case in the Rhapsody-Browser. TestConductor created a witness scenario to display the error. The red arrow visualizes the faulty step and the reason. TestConductor expects the parameter value „OK“, but observes the value „Ready“ during test execution. The expected value was not specified correctly... by accident.

Results	
Status:	FAILED
Progress:	33% (1/3)

Detailed Assertion Information	
'show': Check of in value of argument aMsg (Actual value: "Ready")	FAILED

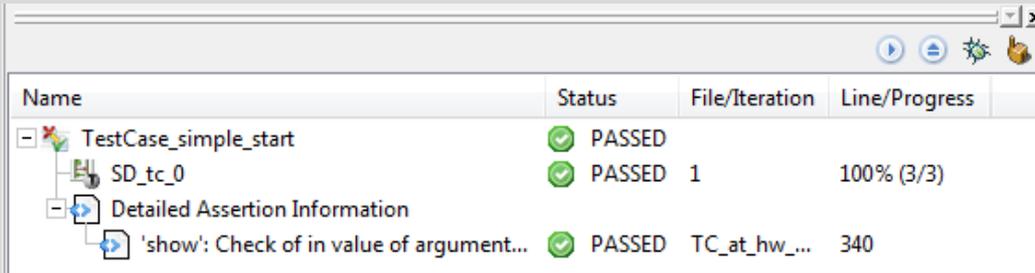
**1** To create and open the witness scenario right click the item SD\_tc\_0 in the TestConductor execution dialog...

**2** ... and select "Show as SD". The witness scenario is added to the model for later inspection.

**3** In the browser, below the test cases, you can find the generated html report. Double click the report to open it.

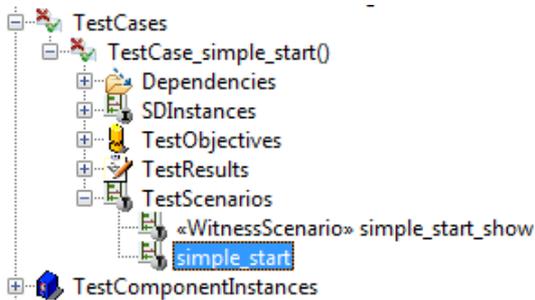
# Test Case Execution III

The test execution PASSED with Rhapsody TestConductor after we corrected the expected parameter value for argument “aMsg” from “OK” to “Ready” in the test scenario „simple\_start“. After changing the scenario, updating, building and re-executing the test case, the test case is passed.

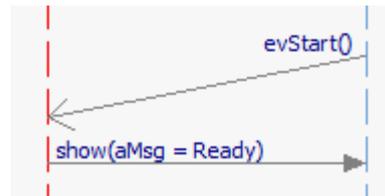


Name	Status	File/Iteration	Line/Progress
TestCase_simple_start	✓ PASSED		
SD_tc_0	✓ PASSED	1	100% (3/3)
Detailed Assertion Information			
'show': Check of in value of argument...	✓ PASSED	TC_at_hw_...	340

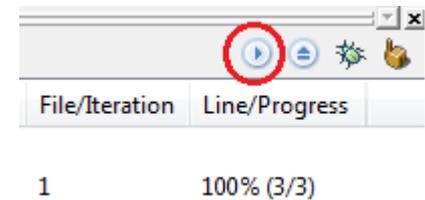
Refer to the user guide to get familiar with the extended functionality of TestConductor.



**1** To correct the test case open the test scenario „simple\_start“.



**2** Respecify the „show“-message parameter value from „OK“ to „Ready“ and close the test scenario.



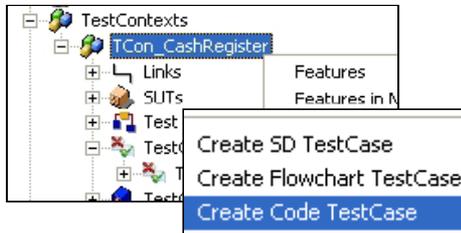
**3** Re-execute the test case by pressing the “Start” button in the top right corner of the execution dialog.

# Test Case: Source Code I

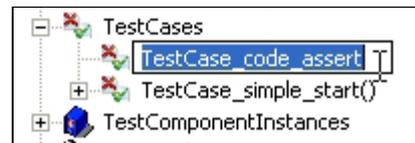


To manually create a source code test case create a code test case and write the test code into the edit field under the implementation tab of the test case. The Rhapsody-TestConductor-macro „RTC\_ASSERT\_NAME“ takes a name-parameter and a condition. If the condition („isNoMoreProducts“) evals to true the test case will pass.

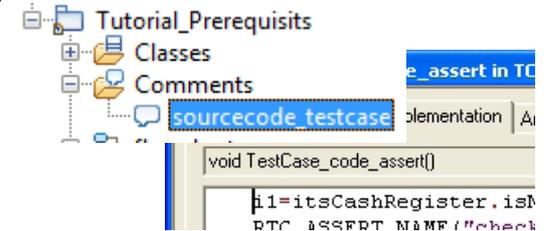
The package “Tutorial\_Prerequisites” contains a comment “sourcecode\_testcase” with the pre defined code for the code test case.



**1** Select the test context „TCon\_CashRegister“ and choose from the context menu „Create Code TestCase“.

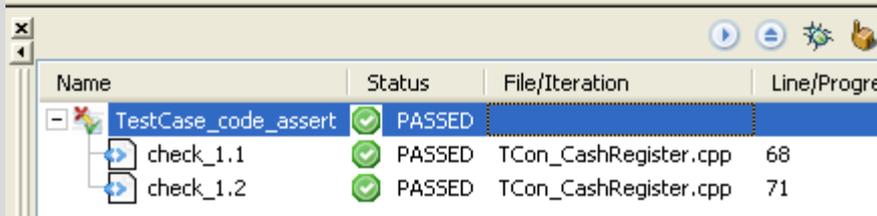


**2** Rename the created test case to „TestCase\_code\_assert“ and open the feature dialog.



**3** Replace the content of the edit field under the implementation tab of the test case with the content from the description field of the comment and press „OK“.

# Source Code Test Case: Execution

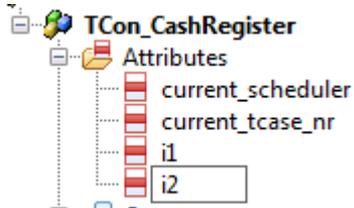


Name	Status	File/Iteration	Line/Progre
TestCase_code_assert	PASSED		
check_1.1	PASSED	TCon_CashRegister.cpp	68
check_1.2	PASSED	TCon_CashRegister.cpp	71

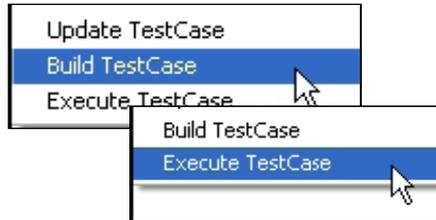
```
i1=itsCashRegister.isNoMoreProducts();  
RTC_ASSERT_NAME("check_1.1", i1==1);  
itsCashRegister.addProduct(new Product(  
i2=itsCashRegister.isNoMoreProducts());  
RTC_ASSERT_NAME("check_1.2", i2==0);
```

Execute the test case with Rhapsody TestConductor.

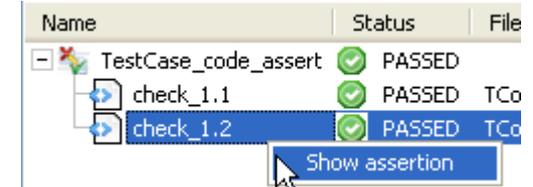
Both assertions evaluate to true and the test case passes. Double-clicking an evaluated assertion in the execution window highlights the assertion in the test model.



**1** Create in the test context „TCon\_CashRegister“ the two integer attributes „i1“ and „i2“.

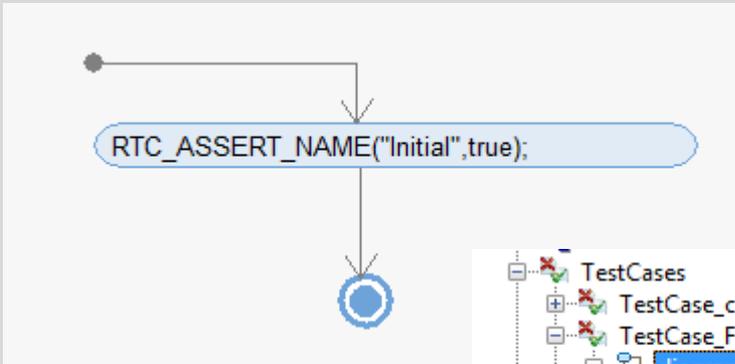


**2** Select „Testcase\_code\_assert“ and choose update, build and execute from the context menu.

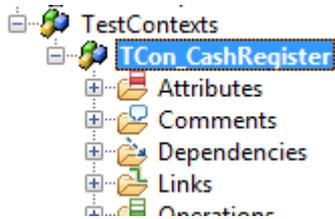
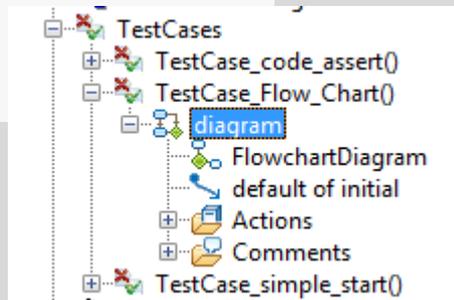


**3** In the execution window, select the assertion and double-click “Show Assertion” in order to highlight the assertion in the model.

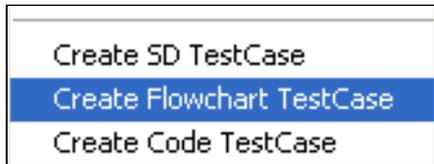
# Test Case: Flow Charts I



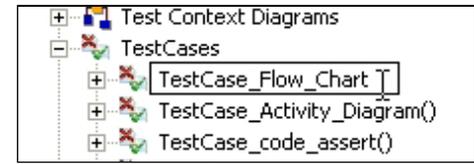
To manually create a flow chart test case we have to define a test scenario which is represented as a flow chart and link it to a test case. TestConductor simplifies this process with a single command.



**1** Select the test context „TCon\_CashRegister“ in the Rhapsody-Browser ...



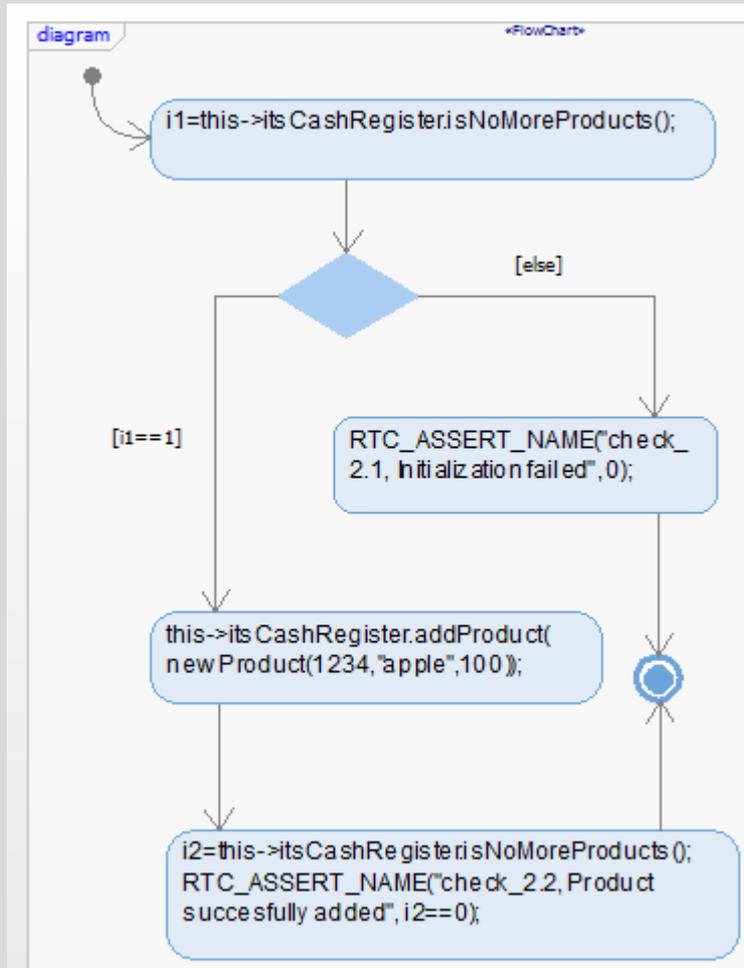
**2** ... and choose from the context menu „Create Flowchart TestCase“.



**3** Rename the created test case to „TestCase\_Flow\_Chart“ and open the flow chart.

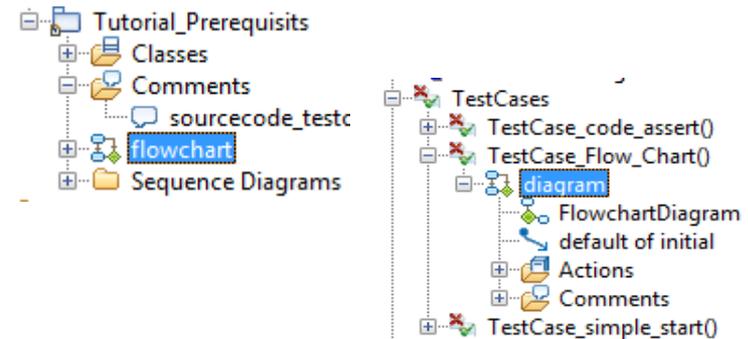
# Test Case: Flow Charts II

2



Define the flow chart in order to execute it with TestConductor. The Rhapsody-TestConductor-macro „RTC\_ASSERT\_NAME“ takes a name-parameter and a condition. If the conditions [i1==1] and [i2==0] evaluate to true the test case will pass.

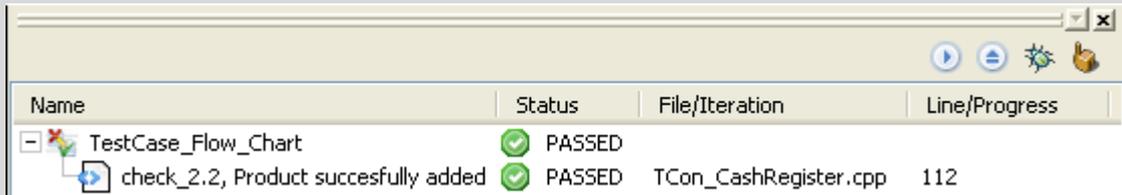
Obviously the flow chart test case is very similar to the source code test case we discussed some pages before. The difference in comparison with the source code test case is the graphical nature of this test case.



1

Replace the content of the flow chart of the test case with the content from the flow chart in Package „Tutorial\_Prerequisites“.

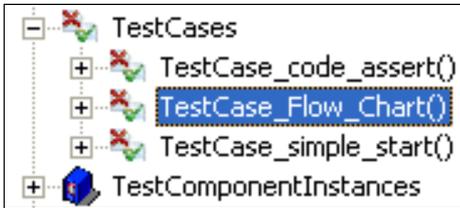
# Flow Charts Test Execution



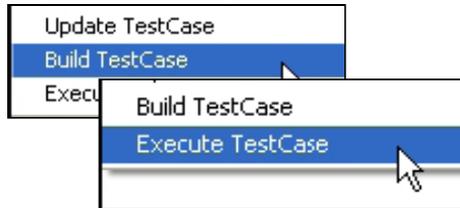
Name	Status	File/Iteration	Line/Progress
TestCase_Flow_Chart	✓ PASSED		
check_2.2, Product succesfully added	✓ PASSED	TCon_CashRegister.cpp	112

```
i2=this->itsCashRegister.isNoMoreProducts();  
RTC_ASSERT_NAME("check_2.2, Product  
succesfully added", i2==0);
```

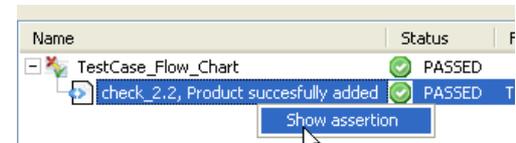
Execute the test case with Rhapsody TestConductor. The „RTC\_ASSERT\_NAME“-macro evals to true and the test case passes.



**1** Select the test case „TestCase\_Flow\_Chart“ ...

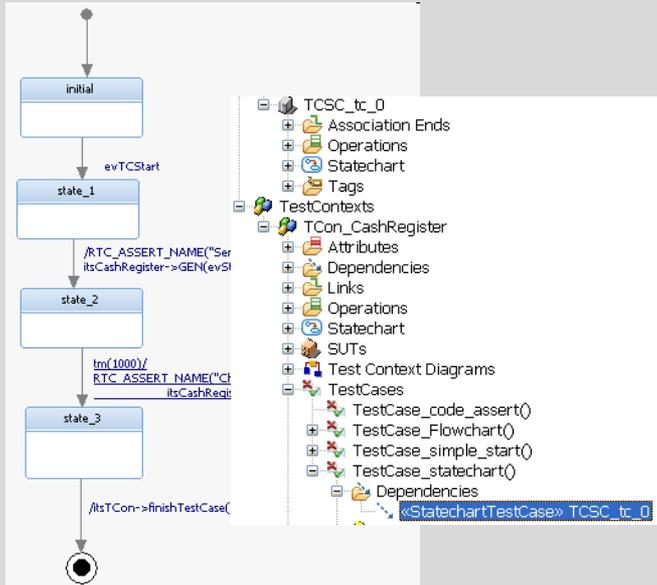


**2** ... and choose from context menu the items „Build TestCase“ and „Execute TestCase“.

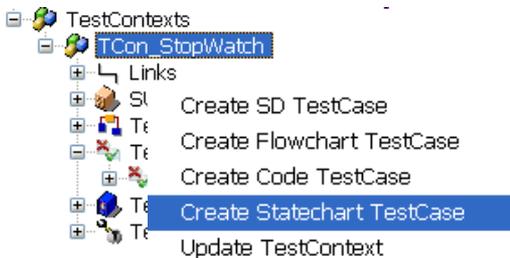


**3** In the execution window, double click on the assertion or right click on it and select “Show Assertion” in order to highlight the assertion in the model.

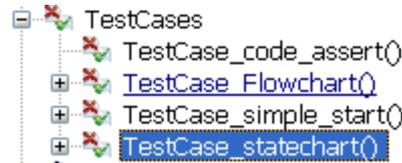
# Test Case: Statecharts I



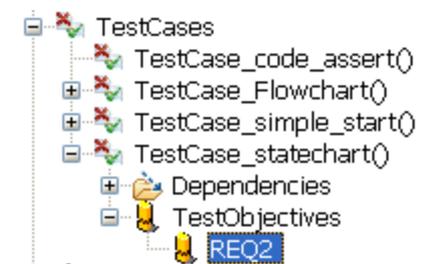
To manually create a statechart test case we have to define a test scenario which is represented as a statechart and link it to a test case. Technically, the test case has a dependency to a TestComponent that contains the statechart. TestConductor simplifies this process with a single command.



**1** Select the test context „TCon\_CashRegister“ and select “Create Statechart TestCase”.



**2** Rename the test case to “TestCase\_statechart”

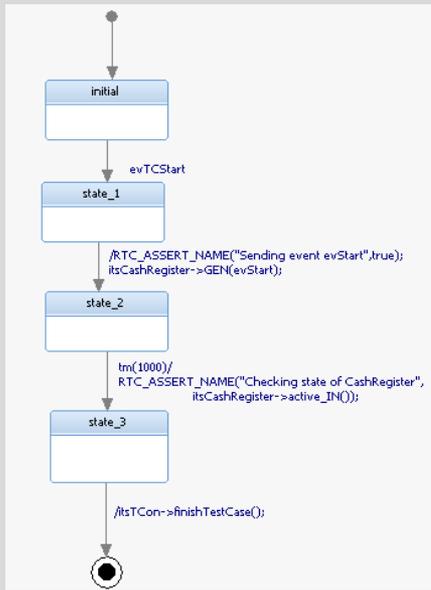


**3** Add a test objective (using “Add New -> TestingProfile ->TestObjective”) to requirement REQ\_2

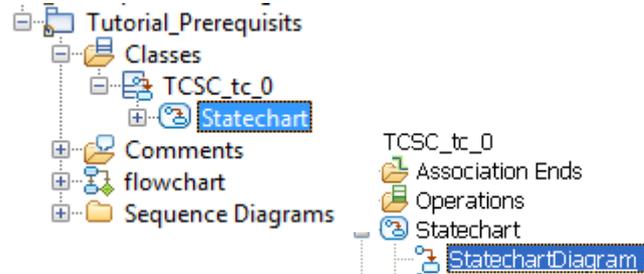
# Test Case: Statecharts II

2

4



Define the statechart in order to execute it with TestConductor. The statechart test case first starts the CashRegister by sending event evStart. After sending this event, the test case waits 1 second. After 1 second has elapsed, the test case checks if the CashRegister has changed its state from idle to active after receiving the event evStart. If both checks are passed, the complete test case is passed.



1

Replace the content of the test component statechart associated with this test case with the statechart of the Tutorial package.



2

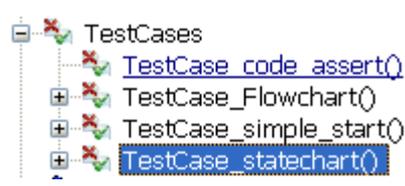
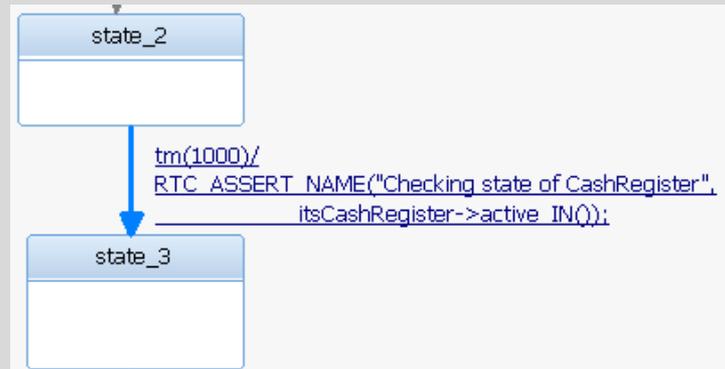
Add "evTCStart" as trigger of the transition from state "initial" to state "state\_1"

# Statechart Test Case Execution

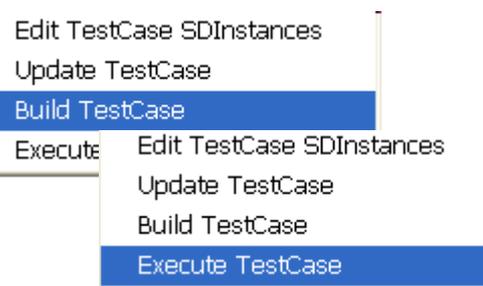
Execute the test case with Rhapsody TestConductor. Both assertions evaluate to true and the test case passes.



Name	Status	File/Iteration	Line/Prog
TestCase_statechart	✓ PASSED		
Sending event evStart	✓ PASSED	TCSC_tc_0.cpp	186
Checking state of CashRegister	✓ PASSED	TCSC_tc_0.cpp	215



**1** Select the test case „TestCase\_statechart“ ...



**2** ... and choose from context menu the items „Update TestCase“, „Build TestCase“ and „Execute TestCase“.

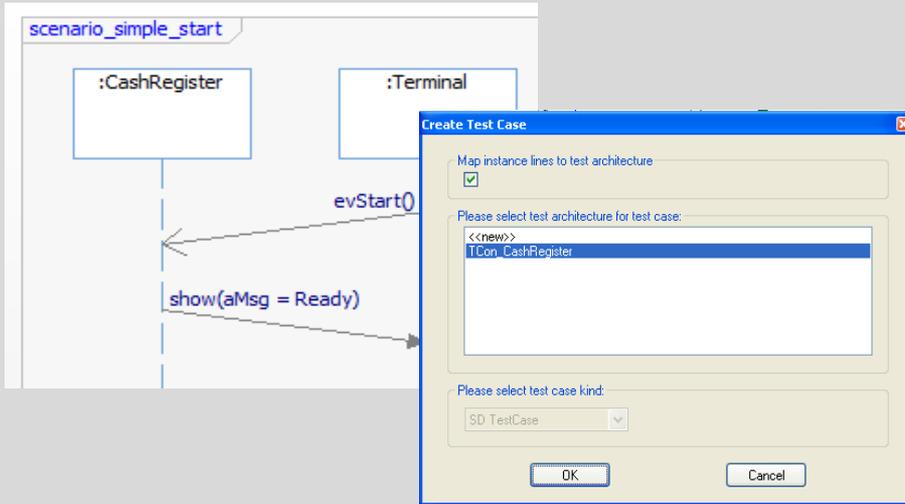


**3** In the execution window, double click on the assertion or right click on it and select “Show Assertion” in order to highlight the assertion in the model.

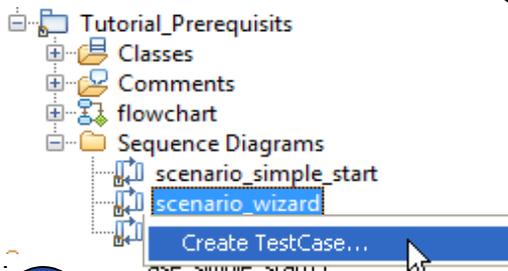
# Create Test Cases Using Test Case Wizard - SDs

2

6



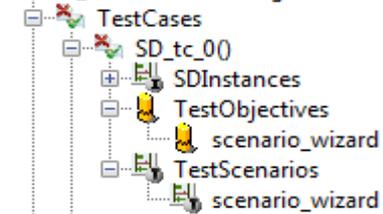
To create a test case based on existing sequence diagrams, operations or requirements, you can use the TestConductor test case wizard. For an existing sequence diagram, the test case wizard creates an analogue test case with the same message structure as the original sequence diagram. For a requirement the test case wizard creates a test case with the chosen requirement as the test objective.



**1** Select the sequence diagram “scenario\_wizard” in the tutorial package and select “Create TestCase...”.

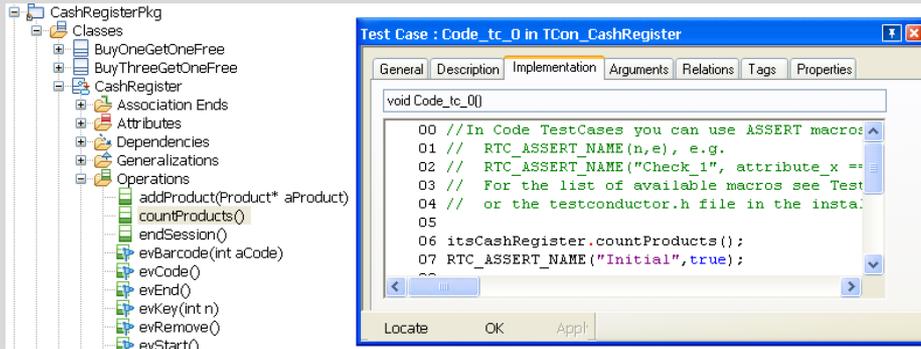


**2** In the test case wizard dialog, the test context “TCon\_CashRegister” is already highlighted. Press OK to proceed.



**3** As a result, a new test case “SD\_tc\_00” has been created which is based on a new test scenario containing the same messages as the original SD, but life lines adapted to the test context structure.

# Create Test Cases Using Test Case Wizard - Operations



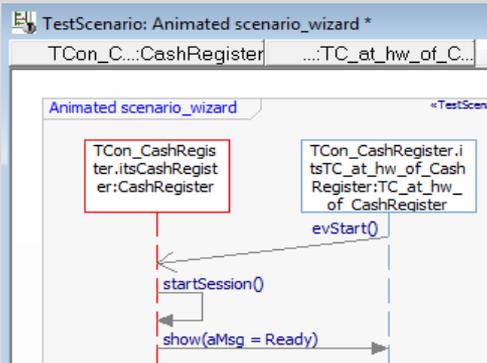
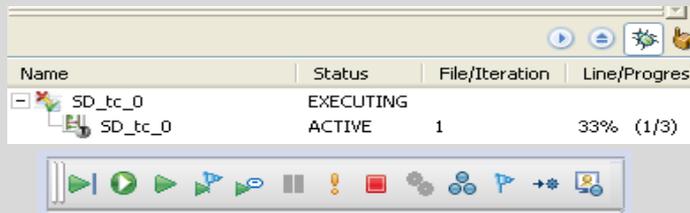
The test case wizard can also be used to test operations that are defined in the model. The wizard allows to create four different kinds of test cases: sequence diagram test cases, statechart test cases, flow chart test cases or code test cases. Independent of the chosen kind of test case, the created test case calls the selected operation. Additionally, the test case already contains a check that can be refined by the user in order to check the out values of the operation.

1 Select operation "countProducts" of class CashRegister in the browser and select "Create TestCase..."

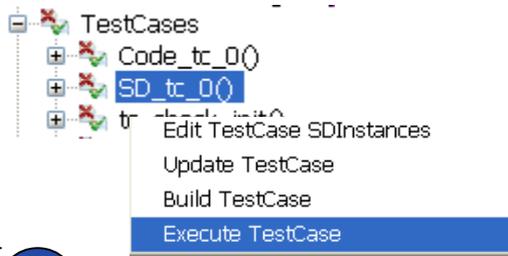
2 In the test case wizard dialog, select "Code Test Case" as test case kind and press OK.

3 As a result, a new code test case has been created that contains a call to operation "countProducts" and also a dummy assertion that can be refined.

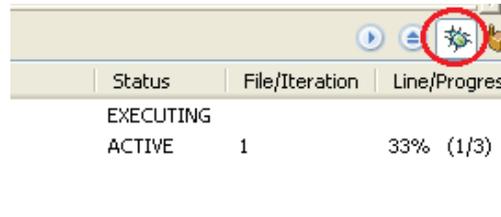
# Debugging Test Cases



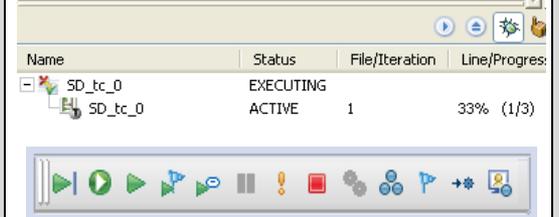
Debugging failed test cases can also be done with TestConductor. When a test case fails, one can turn on debug execution mode in TestConductor's execution window. After switching on debug mode, when executing the test case one can step through it by using the "Go Step", "Go Idle", etc. buttons of Rhapsody's animation toolbar. Additionally, when stepping through the test case, one can use Rhapsody's animation features to inspect animated statecharts, animated SDs, etc. in order to find the reason why the test case fails. In this mode, the application is not terminated automatically after the test case has ended.



**1** Select test case "SD\_tc\_0" and select "Execute TestCase".



**2** After the test case has failed, turn on debug execution mode by clicking the debug button in the execution dialog.



**3** Execute the test case again by pressing the "Start" button in the execution dialog. Now you can step through the test case by using Rhapsody's animation toolbar.

# Executing Multiple Test Cases

Executing multiple test cases can be done by executing a complete test context or a complete test package. When a test context or a test package is executed, all test cases within the context or test package are executed. After all test cases have been executed, TestConductor computes an overall test result for the test context or the test package.

Name	Status	File/Iteration	Line/Progress
TCon_CashRegister	FAILED		
Code_tc_0	PASSED		
Initial	PASSED	TCon_Cas...	86
SD_tc_0	FAILED		
SD_tc_0	FAILED	1	66% (2/3)
TestCase_code_assert	PASSED		
check_1.1	PASSED	TCon_Cas...	135
check_1.2	PASSED	TCon_Cas...	138
TestCase_Flow_Chart	PASSED		
check_2.2, Pr...	PASSED	TCon_Cas...	112
TestCase_simple_start	PASSED		
SD_tc_1	PASSED	1	100% (3/3)

Test Context: TCon_CashRegister	Summary: FAILED
Code_tc_0	PASSED
SD_tc_0	FAILED
TestCase_code_assert	PASSED
TestCase_Flow_Chart	PASSED
TestCase_simple_start	PASSED

**1** Select the test context "TCon\_CashRegister" and select "Update TestContext". After that, select "Build TestContext".

**2** Select the test context again and press "Execute TestContext". All test cases will be executed one after the other.

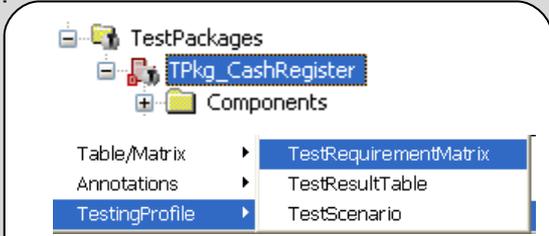
Name	Status
TCon_CashRegister	FAILED
Code_tc_0	PASSED
Initial	PASSED
SD_tc_0	FAILED
SD_tc_0	FAILED

**3** The results are shown in the execution window. As always, "Show as SD" resp. "Show assertion" can be used to show the reasons of failed test cases.

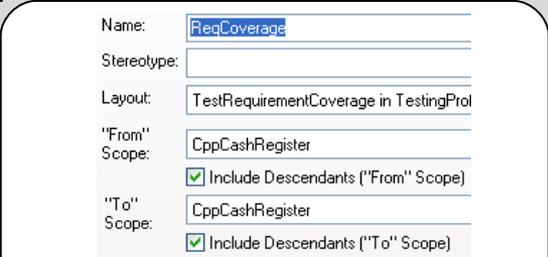
# Assessing Test Case Requirement Coverage I

Which requirements are covered by my test cases? This important question can be answered either by using a test case requirements matrix or by generating a requirements coverage test report. A test case requirements matrix shows the relationship between test cases and requirements in a matrix view. A requirements coverage test report shows the same information, but presented as a textual report. It can be generated by ReporterPlus using a predefined template.

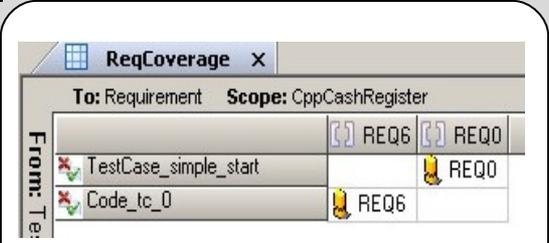
ReqCoverage x	
To: Requirement	Scope: CppCashRegister
From: Test Case	REQ1 REQ2 REQ3 REQ4 REQ5 REQ6 REQ7 REQ8 REQ9 REQ10
TestCase_simple_start	
TestCase_code_assert	
TestCase_Flow_Chart	
Code_tc_0	REQ6
SD_tc_0	



**1** Select the test package "TPkg\_CashRegister" and select "Add New -> TestingProfile -> TestRequirementMatrix".



**2** Open the features dialog of the matrix, rename it to "ReqCoverage", and set the "from" scope and the "to" scope to the complete model "CppCashRegister".



**3** When double clicking the matrix in the browser, the matrix view shows the relationship between the test cases and the requirements.

# Assessing Test Case Requirement Coverage II

3

## All Requirements

Name	ID	Covered By Test Case
<a href="#">REQ1</a>	REQ1	<ul style="list-style-type: none"><li>not covered</li></ul>
<a href="#">REQ2</a>	REQ2	<ul style="list-style-type: none"><li><a href="#">TestCase_statechart</a></li></ul> <p>in TPkg_CashRegister::TCon_CashRegister_Archite</p> <p><b>PASSED</b> for CG Configuration TPkg_CashRegister::TCon_CashRegister_Architec</p>
<a href="#">REQ3</a>	REQ3	<ul style="list-style-type: none"><li>not covered</li></ul>

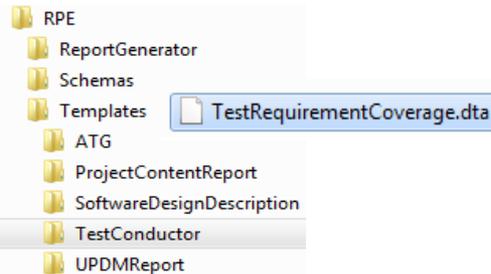
**Test Case Reports** can be used as an alternative in order to figure out coverage of requirements with the test cases. With Rational Publishing Engine a requirement coverage report can be generated in different formats like Word, Html, etc. The requirements coverage report shows the list of requirements, their coverage by test cases and the outcome of the test case execution. The report also contains information about the specification of the test cases.

### Rational Publishing Engine

Generate report...

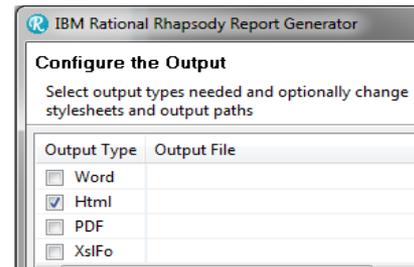
1

From Rhapsody's tools menu, select "Rational Publishing Engine"  
-> "Generate Report..."



2

Select the "TestRequirementCoverage.dta" as template for the report to generate and click Next in the following dialogs.



3

Select the desired output format, html for example, and click on Finish. After generating the report, the report can be viewed with any browser that can display Html files.

# Assessing Test Case Model Coverage

3

2

Detailed Coverage Summary of CashRegister (9/25)

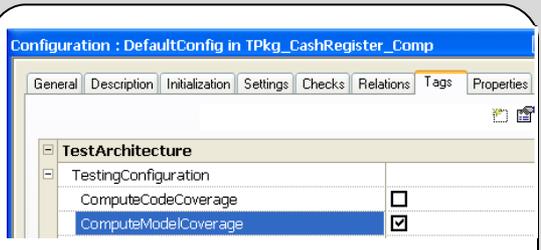
Operations	
not covered	<a href="#">identifyProduct</a>
covered	<a href="#">addProduct</a>
covered	<a href="#">startSession</a>
not covered	<a href="#">endSession</a>
not covered	<a href="#">generateTicket</a>
covered	<a href="#">isNoMoreProducts</a>
not covered	<a href="#">removeLastProduct</a>
covered	<a href="#">countProducts</a>

EventReceptions	
covered	<a href="#">evStart</a>
not covered	<a href="#">evBarcode</a>
not covered	<a href="#">evEnd</a>

Click to highlight element in Rha

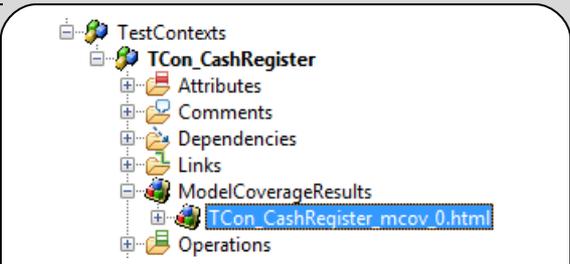
Besides coverage of the requirements, an important orthogonal information is which parts of the model are executed by the test cases, i.e, what is the achieved **Model Coverage** when executing the test cases. TestConductor can compute this information during test case execution. When model coverage computation is turned on, after test case execution TestConductor adds a model coverage report to the test cases, test contexts etc. that shows the achieved model coverage.



**1** On the tags tab of the configuration, turn on "ComputeModelCoverage".



**2** Execute the test context "TCon\_CashRegister".



**3** After execution has finished, model coverage reports can be found both for individual test cases as well as a cumulative coverage report for the test context.

# Assessing Test Case Code Coverage I

3

## 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%

Besides coverage of the requirements and model elements, an important additional information is to what extent the code of the SUT generated by Rhapsody's code generator is executed, i.e, which **Code Coverage** is achieved when executing the test cases. TestConductor can compute this information during test case execution. When code coverage computation is turned on, after test case execution TestConductor adds a code coverage report to the test cases, test contexts etc. that shows the achieved code coverage.



1

Create a copy of the Rhapsody code generation configuration "DefaultConfig", rename it to "Release" and make it the active configuration.



2

On the tags tab of the configuration, turn off "ComputeModelCoverage" and turn on "ComputeCodeCoverage".

# Assessing Test Case Code Coverage II

**Coverage Report**

Environment Info    Table Of Contents    Global Statistics    Source Code

```
33b      {
34      cleanUpRelations();
35      }
36
37  --> 37  bool CashRegister::hw_C::InBound_C::send(I0xFEvent* event, const I0xFEventGenerationParams& params)
37b      {
38      bool res = false;
39      if (event != (0))
39b      {
40      event->setPort(getPort());
41      if (itsIBarcodeReader != (0))
41b      {
42      if (event->isTypeOf(24601))
42b      {
43      res = itsIBarcodeReader->send(event, params);
44      return res;
45      }
46      if (itsIKeyboard != (0))
47b      {
48      if (event->isTypeOf(24602))
48b      {
49      res = itsIKeyboard->send(event, params);
50      return res;
51      }
52      if (event->isTypeOf(24604))
52b      {
53      res = itsIKeyboard->send(event, params);
54      return res;
55      }
55      }
```

Line	Covered	Percentage
43	43	61.4%
1	1	16.7%
0	0	n.a.
7	7	35%
7	7	35%

Source Code

Environment Info

Statement Coverage

Decision Coverage

Condition Coverage

Modified Condition/Decision Coverage

The Code Coverage report contains detailed information to what extent the code of the SUT has been executed by the test cases. The report contains both a summary about the achieved coverage (e.g. statement coverage) as well as detailed information about each single line of code. The source code view contains color coded presentations about the coverage status of statements, decisions and conditions of the tested code.

1 On the settings tab of the configuration, set Instrumentation Mode to "None".

Instrumentation Mode: None

2 Select the test context again and do "Update TestContext", "Build TestContext" and then "Execute TestContext".

TestContexts

- TCon\_CashRegister
  - Attributes
  - Build TestContext
  - Execute TestContext
  - Update TestArchitecture

3 After test case execution has finished, by double clicking the code coverage element in the browser you can open the code coverage report.

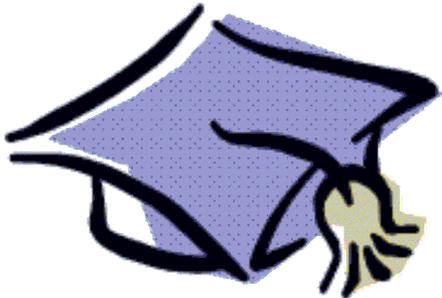
TCon\_CashRegister\_Architecture

- Dependencies
- TestComponents
- TestContexts
  - TCon\_CashRegister
    - Attributes
    - CodeCoverageResults
      - COV

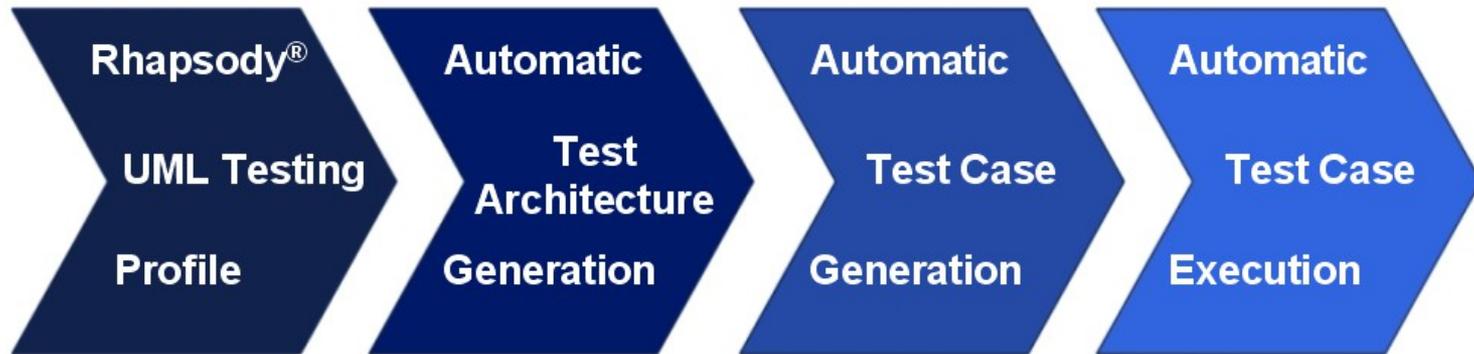
# Conclusion

## The high-grade automation in the Rhapsody Testing Environment with TestConductor

- /// generates complete, immediately executable test architectures in shortest time with a few mouse clicks.
- /// makes it for the first time possible to implement cyclically quality assurance measures in early phases of the development.
- /// increases substantially the planning reliability for projects, because design errors and subsequent errors will be recognized very early.
- /// makes statements about the coverage rates for both the model elements and model code. Developers can easily and fast analyze reasons for not covered elements.
- /// highly automates the testing process and **can save test development time** compared to traditional approaches.



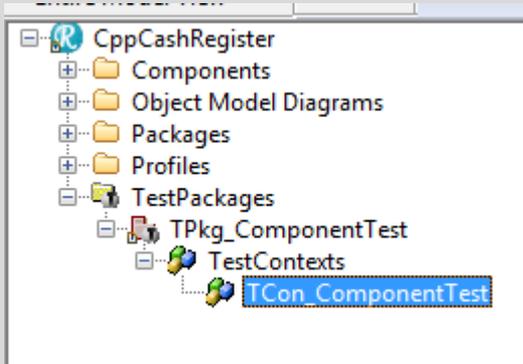
## Testing a Rhapsody Component!



*Rhapsody® Testing Environment*

# Generate Test Architecture

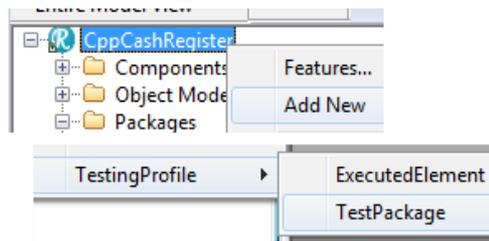
3



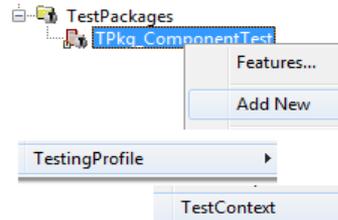
To manually create a test architecture

for the component test, insert a new test package and a new test context. It is not necessary to define a SUT and test components. We will use the pre-defined component «CashRegisterNoGui» and its configuration «Debug»; activate this configuration before you proceed. This test validates the complete model running in a production configuration against its requirements. Here, the SUT is the complete component.

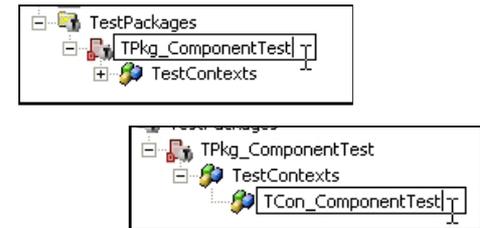
Important is, this is possible only for animation based testing mode (see below).



- 1 Select the root package „CppCashRegister“ and choose from the context menu „Add New -> TestingProfile->TestPackage“. Then open the Features dialog of the TestPackage and set the property „TestConductor::Settings::Testing Mode“ to „AnimationBased“.

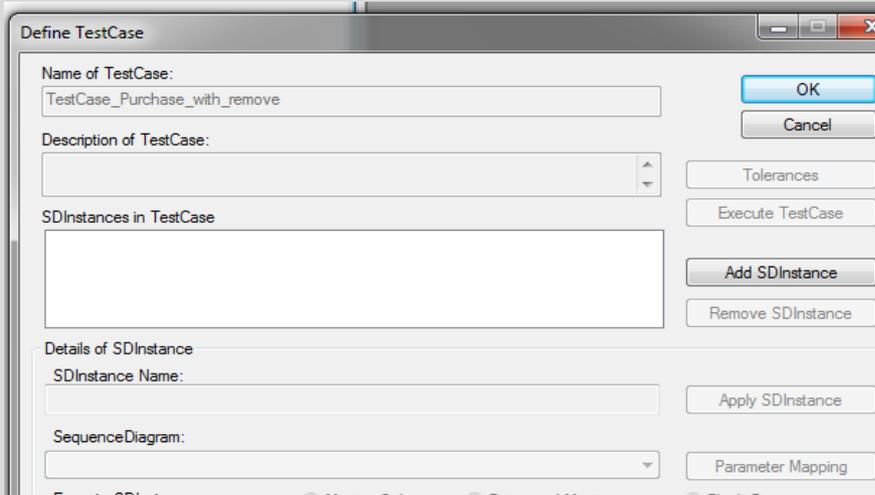


- 2 Select the created test package and choose from the context menu „Add New -> TestingProfile -> TestContext“.



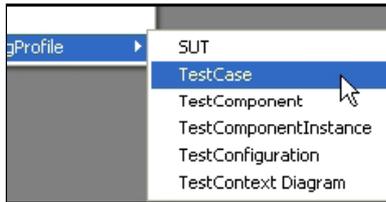
- 3 Rename the created test package to „TPkg\_ComponentTest“ and the created test context to „TCon\_ComponentTest“.

# Link SD to Test Case

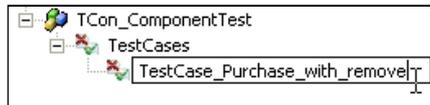


To link an existing sequence (requirement) diagram to a test case create a test case and open the dialog „Define Test“.

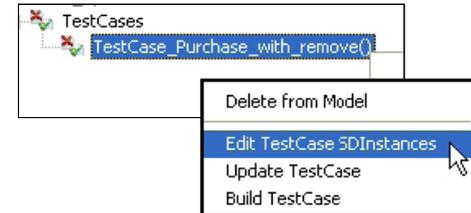
In the dialog „Define Test“ the user can specify properties concerning the execution of sequence diagram test cases. Refer the user guide to get familiar with the properties and their effect during test case execution.



**1** Select the test context „TCon\_ComponentTest“ and choose from the context menu „Add New -> TestingProfile -> TestCase“.



**2** Rename the created test case to „TestCase\_Purchase\_with\_remove“.

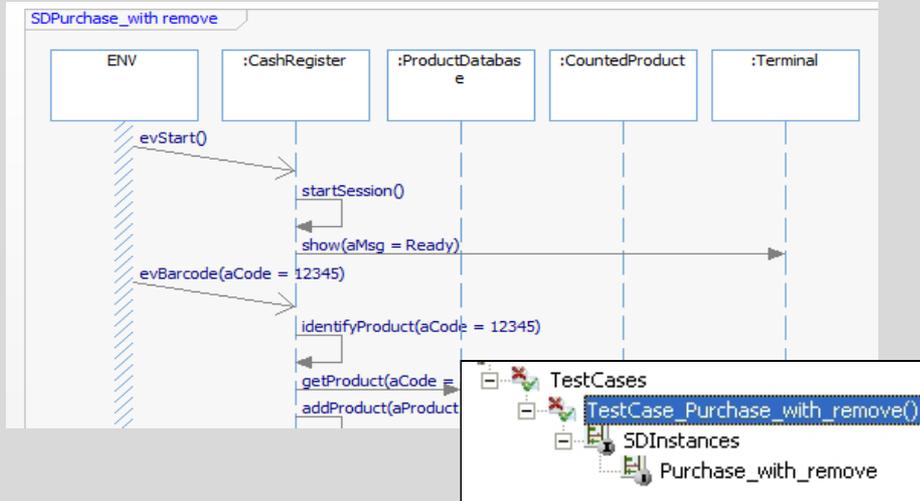


**3** Select the test case „TestCase\_Purchase\_with\_remove“ and choose from context menu the item „Edit TestCase SDInstances“.

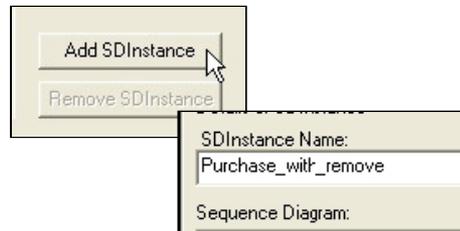
# Test Case Property Definition

3

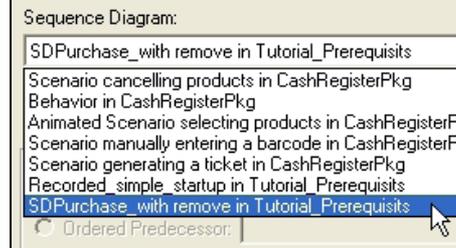
9



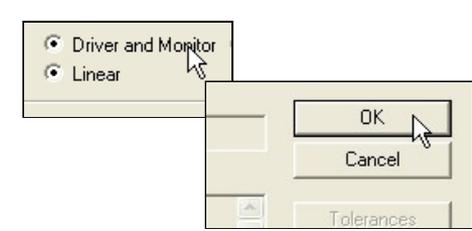
Define the properties of a test case in order to use an existing sequence diagram. In the dialog „Define Test“ specify the sequence diagram, switch to linear driving and apply the changes. We use the sequence diagram „SDPurchase\_with\_remove“ from the specification phase of the CashRegister project, which specifies a complete purchase process.



**1** Press „Add SDInstance“ and write in the field „SDInstance Name“ the text „Purchase\_with\_remove“.



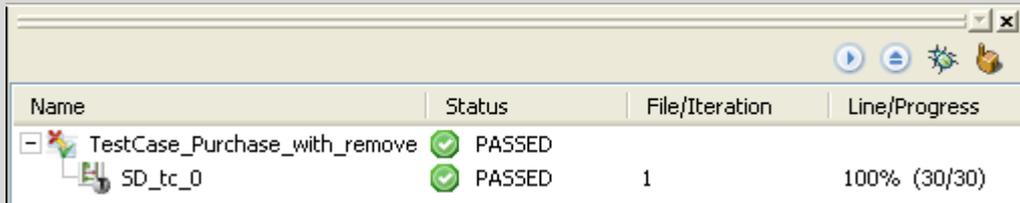
**2** Select the item „SDPurchase\_with\_remove“ from the drop-down combobox in the field „Sequence Diagram“.



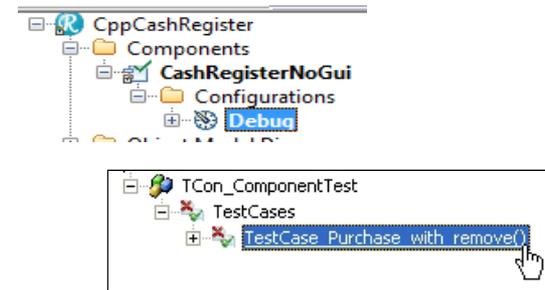
**3** Select the „Driver and Monitor“ option and apply all changes by pressing „OK“. The dialog closes.

# Passed Test Execution

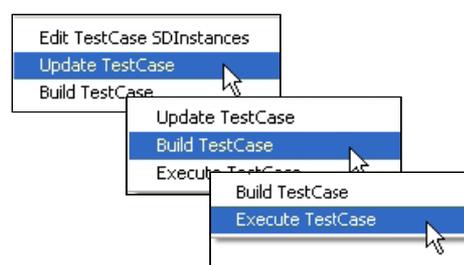
The test execution **PASSED** with Rhapsody TestConductor.



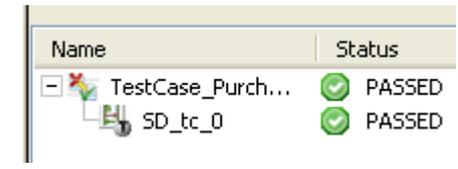
Name	Status	File/Iteration	Line/Progress
TestCase_Purchase_with_remove	✓ PASSED		
SD_tc_0	✓ PASSED	1	100% (30/30)

**1**

Set as active component "CashRegisterNoGui".  
Select the test case "TestCase\_Purchase\_with\_remove".

**2**

... and choose from context menu the items "Update TestCase", "Build TestCase" and "Execute TestCase".



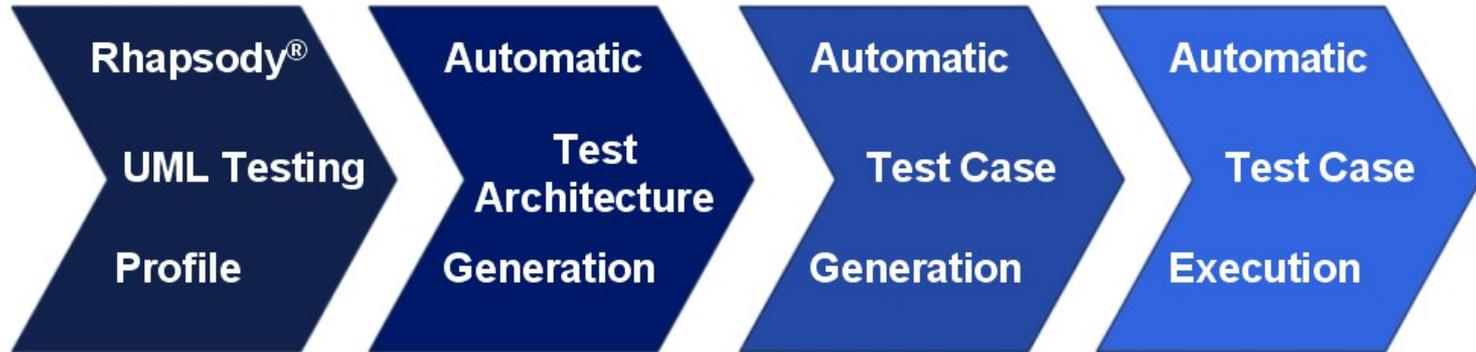
Name	Status
TestCase_Purch...	✓ PASSED
SD_tc_0	✓ PASSED

**3**

The test case runs and passes as expected.

# Appendix II

## Generating test reports with Rhapsody ReporterPLUS!



*Rhapsody® Testing Environment*

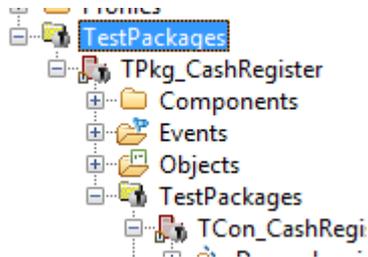
# Test Report Generation I

4



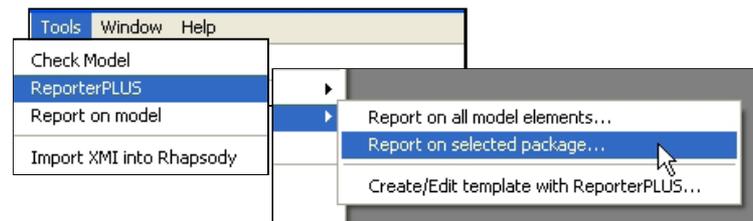
To generate a test report with Rhapsody ReporterPLUS select a test package in the Rhapsody browser and start the ReporterPLUS wizard. After all needed options are selected ReporterPLUS will start to collect information and displays it in a well arranged style in different formats as listed in the figure.

In opposite to the Rhapsody TestConductor HTML Test Result Report every ReporterPLUS template can be customized to fit the users needs.



1

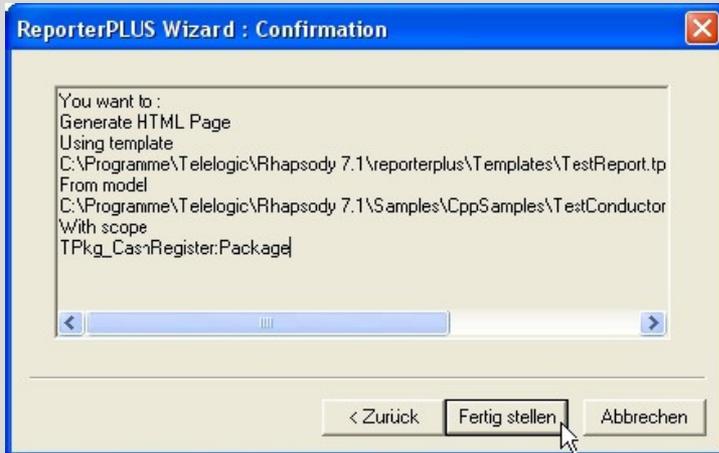
Select the test package „TPkg\_CashRegister“ in the Rhapsody browser ...



2

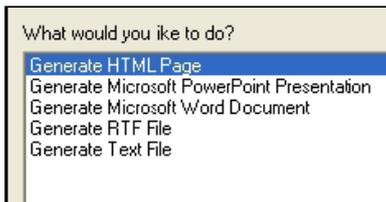
... and choose from the „Tools“ menu „ReporterPLUS -> Report on selected package...“ to create a report for the selected test package. In case a report for all test packages in the model shall be created, choose the menu item „Report on all model elements...“

# Test Report Generation II

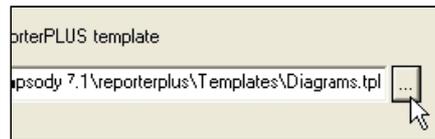


Select the export format and choose the test report template, which has been installed with Rhapsody TestConductor in the ReporterPLUS template directory. This template uses the TestingProfile to provide the underlying stereotypes to generate a document.

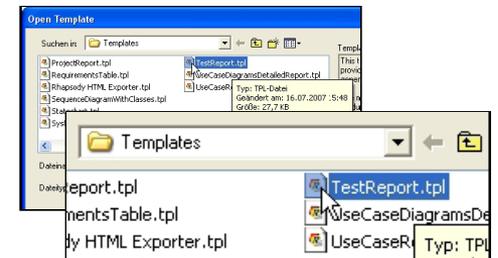
TestConductor provides also a template to generate a test requirement report, TestRequirementCoverage.tpl.



**1** Select the export document format „Generate HTML Page“ and choose „Next>“.



**2** Click on the Button „...“ to browse the test report template.



**3** Select the template „TestReport.tpl“ in the folder „reporterplus\Templates“ in your Rhapsody installation and choose „Next>“.

# Test Report Generation III

**Table of Contents**

- Test Report of Model CppCashRegister
  - TCon\_CashRegister
    - System Under Test (SUT)
    - Test Component Instances
    - Test Context Diagrams
    - Test Cases
      - Test Case AD\_tc\_0
      - Test Case Code\_tc\_0
      - Test Case SD\_tc\_0
        - Scenario SD\_tc\_0
      - Test Case atg\_tc\_002
      - Test Case atg\_tc\_003
      - Test Case atg\_tc\_004
      - Test Case atg\_tc\_006
      - Test Case atg\_tc\_007
      - Test Case atg\_tc\_008
      - Test Case atg\_tc\_009
      - Test Case atg\_tc\_010
      - Test Case atg\_tc\_013
      - Test Case atg\_tc\_014
      - Test Case atg\_tc\_015
      - Test Case atg\_tc\_016
      - Test Case atg\_tc\_017

**Test Report of Model CppCashRegister**  
(Report created at 7/17/2007 at 16:41:04)

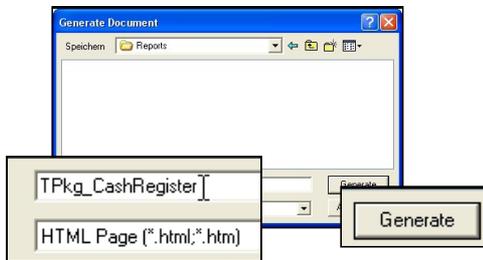
<b>Project</b>	CppCashRegister.rpy
<b>Directory</b>	C:\Programme\Telelogic\Rhapsody 7.1 \Samples\CppSamples\TestConductor\CppCashRegister
<b>Language</b>	C++
<b>Description</b>	This is the CashRegister exercise model for the Rhapsody TestConductor and ATG tutorial. It is based on the model from M.W.Richardson and shows the main aspects of the Testing Profile implementation firstly delivered with Rhapsody TestConductor 2.0.

This document contains the test contexts

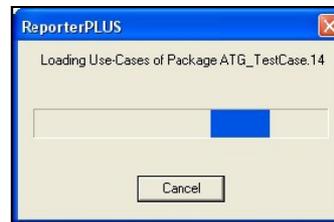
TCon\_CashRegister in TPkg\_CashRegister::TCon\_CashRegister

Specify the report file name and execute Rhapsody ReporterPLUS to display information about the defined parts of your model.

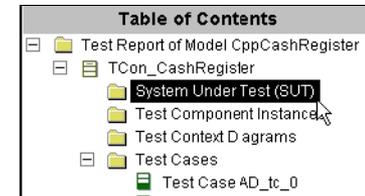
The HTML export format we use for this example needs Microsoft Internet Explorer (or Netscape Navigator) with installed Java virtual machine. In case the virtual machine is not installed, the browser will ask to install it automatically from the internet.



**1** Finish the ReporterPLUS wizard, name the export file in the „Generate Document“ dialog and select „Generate“.



**2** ReporterPLUS will collect information from you model and start the corresponding application for the selected export file format to display.



**3** Discover the browseable information in the report. Select a linked item in the left section to display the corresponding information.

# More Information ...



For further information, especially technical news, visit our internet information portal or contact one of our worldwide sale agencies.

IBM® Rational® Software Support provides you with technical assistance. The IBM Rational Software Support Home page for Rational products can be found at <http://www.ibm.com/software/rational/support/>.

For contact information and guidelines or reference materials that you need for support, read the [IBM Software Support Handbook](#).

For Rational software product news, events, and other information, visit the [IBM Rational Software Web site](#).

Voice support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to <http://www.ibm.com/planetwide>.

Before you contact IBM Rational Software Support, gather the background information that you will need to describe your problem. When describing a problem to an IBM software support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?
- Do you have logs, traces, or messages that are related to the problem?
- Can you reproduce the problem? If so, what steps do you take to reproduce it?
- Is there a workaround for the problem? If so, be prepared to describe the workaround.