# Dev 11 Scenarios for CodeContracts

Authors: Manuel Fahndrich, Mike Barnett

7/1/2010

## History

|  |  |
| --- | --- |
| Date | Description |
| 7/1/2010 | Initial draft |
| 7/7/2010 | Added code exploration/assumption validation scenario (thanks to Mark Mydland) |

Contents

[Dev 11 Scenarios for CodeContracts 1](#_Toc266251182)

[History 1](#_Toc266251183)

[Glossary 1](#_Toc266251184)

[Test Driven Development with Contracts 2](#_Toc266251185)

[Discovering API Rules while Coding 2](#_Toc266251186)

[Narrowing down the Source of Errors with CodeContracts 2](#_Toc266251187)

[What-If Exploration of Code during Code Reviews 3](#_Toc266251188)

## Glossary

* **CodeContracts**: a language independent way of specifying preconditions, postconditions, and object invariants in .NET programs
* **Precondition**: a condition that must be true on entry to a method. Expressed using Contract.Requires.
* **Postcondition**: a condition that must be true on normal exit of a method. Expressed using Contract.Ensures.
* **Contracts**: a term used to refer to preconditions, postconditions, and/or object invariants.
* **Exceptional Postcondition**: a condition that must be true when leaving a method on throwing an exception. Expressed using Contract.EnsuresOnThrow.
* **Object invariant**: a condition on a data structure that should hold on public method exits and at the end of constructors. Expressed using Contract.Invariant in a method annotated with [ContractInvariantMethod].
* **TDD**: Test-Driven-Development. A methodology where programmers write failing tests first, and then write code to make the tests pass (also known as Red-Green Workflow).
* **Pex** is an automated white box data generator for .NET. Given a method, Pex can generate inputs that cover corner cases.
* **Rewriter:** ccrewrite.exe is a CodeContract tool that instruments a .NET assembly after compilation to insert contract checks at appropriate places in the code. It reads contracts from the assembly being rewritten and from Contract Reference Assemblies for referenced dlls. Postconditions are inserted at the appropriate method return points, as well as object invariant checks. All contracts are inherited, so e.g., if an interface has contracts, then implementations of the interface are instrumented to check the contracts. The original condition string for a contract is extracted from the source during rewriting and inserted into the code in order to provide the failing condition at runtime.

## Test Driven Development with Contracts

Matt has to write a TrimSuffix method that removes arbitrary string suffixes from an original string. He uses TDD and writes a set of small tests. At the same time, he also thinks about the generalized outcome of the method and writes a Contract.Ensures to specify that the result should not end in the suffix.

As he is testing with various inputs he refines the assumptions of the method and documents them using Requires.

To see if he has covered all the corner cases, Matt runs Pex on his code. Pex discovers a new way to violate the post-condition that Matt missed, namely when the original string contains a repeated suffix and the code removes it only once.

Matt’s team uses XML doc files and automated documentation generation on all their solutions. On the next build, the documentation automatically includes detailed contracts on how to use the new TrimSuffix method.

## Discovering API Rules while Coding

Phil is using the new TrimSuffix method that Matt just checked in. As he types a call to the method in his code, Intellisense shows him the proper requirements of the parameters by means of the method contracts. This way, Phil discovers right away that the method won’t work if the suffix is empty. Phil thus fixes his code to avoid that corner case before even running his first test.

Phil is never quite sure about the boundary conditions on the length parameter when calling String.Substring. He hits F12 with the cursor on the Substring method and is shown the API for System.String, augmented with Contracts. He determines that it is permitted to extract 0 characters at the end of the string.

## Narrowing down the Source of Errors with CodeContracts

Jean is the new member on the team and tasked with maintaining a web site using ASP.NET with code-behind and Silverlight in the browser. Jean has to perform some important refactoring but she is not sure whether her changes are permissible. One of the regression tests now fails, but it is not clear why the result is not what is expected by the test. Fortunately, the team has been using contracts liberally in the Silverlight app and in the code-behind part and Matt shows Jean how to enable the runtime checking of contracts in both the Silverlight projects and the ASP.NET web server projects. When running the tests in Debug mode with contracts instrumented into the product several tests now fail with an object invariant violation at the end of a method Jean modified. Looking at the original invariant, Jean sees exactly what condition she failed to maintain. The failing invariant pinpointed by CodeContracts leads Jean to the source of the error and helps her figure out how to fix the problem, whereas the failing test outcome at the end of the test run left her clueless.

## Exploring Assumptions and Understanding of Code using Contracts

Mary is tasked to change a feature requiring some substantial changes to existing code. Mary has been using CodeContracts in her new code and wants to see if they help her understand existing code that has not made heavy use of CodeContracts. As she tries to understand the existing code with respect to the changes she needs to do, she formulates a model and a set of assumptions in her head about how the code works and what kind of invariants it maintains. As she makes these assumptions, she writes them directly into the code as Contract.Requires, Contract.Ensures, Contract.Invariant, and Contract.Asserts. Using continuous testing on all test suites, she can validate which of her assumptions are wrong and which of them seem to hold over all the test suites.

Given this set of explicit contracts in the code, Mary feels much more confident in her understanding of its current working behavior. These assertions also let her make her changes with more confidence, as they may catch violations of existing invariants introduced by her changes.

## What-If Exploration of Code during Code Reviews

Jean is ready to check-in the feature change she was tasked with. Visual Studio presents her with a ranked list of warnings newly introduced, changed, or fixed by *her changeset*. Jean is not comfortable enough addressing all the warnings herself, so she submits her changeset for review to Matt and Phil.

Matt and Phil see the list of affected warnings in addition to the regular code review data. Matt suggests some extra preconditions and a few code changes to avoid potential null-dereferences among the warnings. The code review system performs an update of the warning list based on Matt’s suggested preconditions and other changes. As a result, a call-site is found that violates the new preconditions, which causes Matt to file a bug against that piece of code.

Phil investigates the remaining warnings and classifies them as benign or not likely.

After changing the code and submitting the update for the review, no new unclassified warnings are reported by the system and everyone signs off on the review.