Efficient test case generation

Reliable Software and Architecture Project

Søren Trudsø and Kenneth Egholm

Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark

Group #04

March, 19. 2010

Abstract

We intend to show that using automatic test generation tools make it possible to achieve the same test case quality in less time – compared to a traditional approach.
In this paper we are comparing two very different ways of generating test-cases; Equivalence class partitioning combined with boundary value analysis against using PEX – an automatic white box test generation tool from Microsoft research. Lastly we try to give a recommendation of best practice.

Index

1 Motivation

Testing has become a very important part of the software development process. It is often a natural part of the process and many companies does not accept production code that has not been covered by test cases. As important as writing test cases is it is still natural to ask the question: “can we get to the needed quality of test cases by less effort”?

The motivation of this paper is to evaluate whether it is possible to use a test case generation tool to reduce the time spent on test case generation without compromising the quality of the test cases. At the same time we think testing is a very interesting aspect of the job as software developers, so we jumped at the chance to explore further ways of performing this art.

2 Hypothesis

It is our hypothesis that it is possible – by using an automatic test generation tool – to reduce the time that is spent creating test cases without compromising the quality of the test cases.

3 Method

3.1 Metrics used in evaluation

It is always a daunting task to discuss measuring of quality. In order to do so, we need to establish some definitions as well as some metrics. We have found inspiration for doing this in [Pfaller 2008].

3.1.1 What are we comparing

The object of measurement is in this case the test suites that have been generated from the two different approaches.

3.1.2 Relative quality

As this is a comparison of two approaches to test case generation against a very small specification and therefore also a very small unit of production code it should be noted that the quality we measure here is by nature only relative. Against other kind of production code or other specifications results may be different.

3.1.3 Defects detected

The ability to detect defects in the production code is of course the most important property of a test suite.

3.1.4 Time spent on test case generation

This metric is concerned with the time that is spent with the test case creation. In the case of the traditional approach this includes the equivalence class partitioning, boundary value analysis and the writing of the concrete test cases.

In case of the automatic test case generation tool we decided not to measure time to learn using the tool as mastering the EQ+BV analysis also has a learning curve.

The time metric is not directly related to the quality of the test cases but it is needed to evaluate the two methods against each other.

3.1.5 Code coverage

Code coverage in itself is not very important although it is a useful metric to evaluate the efficiency of the approach [PEX tutorial].

We measure the code-coverage of the two approaches as well as the test suite efficiency. The test suite efficiency is a metric that we have invented ourselves in an attempt to compare the efficiency of two techniques. Efficiency in this case is to have full code coverage in as few test cases as possible.

Test suite efficiency (code coverage)

In addition to measure the code coverage, we have decided to measure the efficiency of the test suite in relation to code coverage. The calculation simply tries to give a measure of how many test cases the suite needs to give a (theoretically) code coverage of 100%.

We will calculate the efficiency like this:

$$number\ of\ testcases * \left(\frac{100}{code\ coverage}\right)$$

Example: a test suite with 25 test cases and realized code coverage of 95% will have an efficiency count of approximately 26.3. Lower is of course better.

This way of measuring efficiency is only meaningful when comparing test suites with a code coverage that is relatively close to each other.

3.1.6 Maintainability of test code

This is a general discussion of maintainability of test code.
Maintainability of a software product is defined as its capability to be modified [ISO SW Quality].

This is in particular an important property of test case suites as it is a costly task to modify existing test cases as production code changes. So in order to compare two test case generation techniques against each other it makes good sense to try to assess the maintainability of the resulting test case suites.

In order to make this an objective and measurable metric, we subdivide it into the following categories.

- Maintainability compliance – Naming of test cases
- Analyzability – How easy is it to identify what sections should be modified;
- Changeability – How much effort is needed to maintain the test suite

Each test suite will be given a grade between 1-5. Highest score is best.

Maintainability compliance

Defining maintainability compliance

It is very important to name test cases by naming conventions/standards. If a certain naming convention is used it is possible to tell a lot about what the test case is testing and what the expected outcome of the test case is. [Osherove 2009] “7.3.1 Naming unit tests” and [Meszaros 2007]. These naming conventions suggest that the name of test cases should consist of three parts:

- The name of the method being tested
- The state and input with which it’s being tested
- The expected behavior

i.e.:

```
public void AnalyzeFile_FileWith3LinesAndProvider_ReadsUsingProvider()
```

Analyzability

How well are the test cases structured so we can identify which section needs to be modified?

Test cases should follow a strict pattern of three parts:

- Arrange
- Act
- Assert

(AAA [Oshorove 2009] or Four Phases Test [Meszaros 2007])
i.e.:

```
public void IsValidFileName_validFile_ReturnsTrue()
{
//arrange
LogAnalyzer analyzer = new LogAnalyzer();
//act
bool result = analyzer.IsValidLogFileName("whatever.slf");
//assert
Assert.IsTrue(result, "filename should be valid!");
}
```

Also other principles apply,

- Isolated Test/Keep tests Independent
- Simple test code (Evident Tests/Communicate Intent)
- Who verifies our output from UUT Evident data/Test as documentation [Meszaros 2007] Chapter 3.
- Verify One Condition per Test

[FRSE 2010] and [Meszaros 2007] Chapter 5.

Changeability

The changeability property of a test suite is concerned with the ability of the test suite to have a specified change implemented.

The changeability of the test suites will be evaluated from the following sub metrics:

- Behavioral changes of the production code
- Refactoring changes of the production code
- Interface changes of the production code

3.2 Production code and test case generation

We are testing the hypothesis by finding a suitable specification which we intend to implement.

This implementation is then subjected to the two different methods of test generation:

- Equivalence partitioning, boundary value analysis and test generation
- Test generation by an automatic test generation tool for explorative testing and regression testing.

The time spent on both methods is measured.

We will conduct experiments with defect seeding and measure code coverage and other metrics for each experiment.

4 Analysis

4.1 The specification

The problem domain (the specification) that we have decided to use is the Danish law “Registreringsafgiftsloven” [LAW].

This law has sufficient complex algorithm to support our purpose. We simplify the specification somewhat, as we confine the problem domain to treat only “normal vehicles”.

This means that we do not calculate special vehicles such as motorcycles, trucks or electric powered vehicles.

The following chapter is our reduced specification from the law:

4.1.1 Specification abbreviation

These are the input variables for the algorithm:

- The price (without VAT) of the car from the dealer
- BaseAmount DKK 79.000
- Rate below BaseAmount: 105%
- Rate above BaseAmount: 180%
- Diesel/Gaz – Has an impact on calculation of the reduction
- Reduction/addition to the tax:
  - Diesel Particle filter -3500
  - Seat belt alarms (Max 3) -200

km/liter(mileage)	more than km/liter	less than km/liter
Gaz 16 km/liter	-4000	+1000
Diesel 18 km/liter	-4000	+1000

4.1.2 The core algorithm of the production code:

The body of the method under test, before running the test suite generated by EQ+BV:

```
var priceWithVAT = priceBeforeTaxWithoutVAT * (1 + VAT);
double registrationTax = 0;
if (priceWithVAT <= BASE_AMOUNT) {
    registrationTax += priceWithVAT * TAXRATE_BELOW_BASE_AMOUNT;
}
else {
    registrationTax += BASE_AMOUNT * TAXRATE_BELOW_BASE_AMOUNT;
    registrationTax += (priceWithVAT - BASE_AMOUNT) *
TAXRATE_ABOVE_BASE_AMOUNT;
}

var mileageLimit = fuel == Fuel.Diesel ? MILEAGE_LIMIT_DIESEL :
MILEAGE_LIMIT_GAZ;
var mileAgeAboveLimit = mileage - mileageLimit;
if (mileAgeAboveLimit > 0) {
    registrationTax += mileAgeAboveLimit * OVER_MILEAGE_RATE;
}
else {
    registrationTax -= mileAgeAboveLimit * UNDER_MILEAGE_RATE;
}

if (countOfAlarms > 0) {
    int alarmsUsedInCalculation = countOfAlarms;
    if (alarmsUsedInCalculation > MAX_COUNT_OF_ALARMS) {
        alarmsUsedInCalculation = MAX_COUNT_OF_ALARMS;
    }
    registrationTax += alarmsUsedInCalculation *
COUNT_OF_ALARMS_RATE;
}

if (fuel == Fuel.Diesel && particleFilter) {
    registrationTax += PARTICLE_FILTER_RATE;
}

return registrationTax + priceWithVAT;
```

c# source code

4.2 How we conduct the experiments

4.2.1 The tools

- Visual Studio 2010 Ultimate RC (VS2010)
  - MSTest - Testing framework
- PEX 0.23 (PEX)
  - Is a Microsoft® Visual Studio® add-in that provides a runtime code analysis tool for .NET Framework code.
- NCover - Used for measuring coverage

VS2010

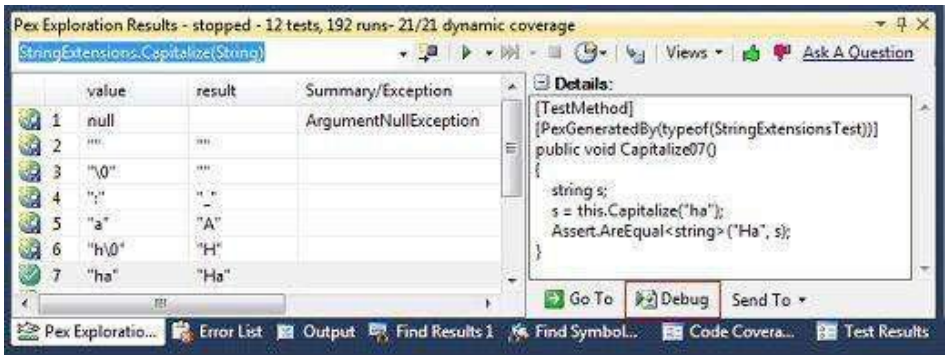
Visual Studio 2010 was used to implement the production code in the .net framework and C#. Writing the test suites and for "hosting" PEX. It was of course also used for running the test suites.

PEX

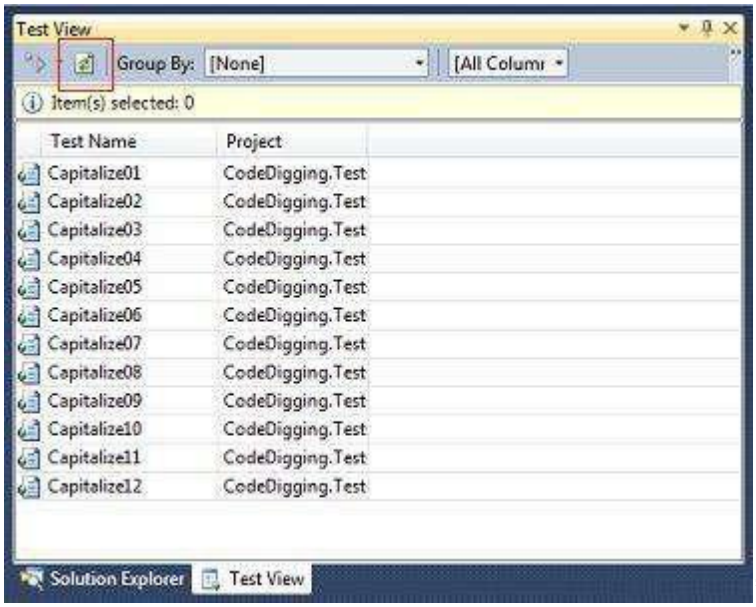
PEX was used for creating parameterized unit test (PUT).
Parameterized unit tests (PUTs) is a new methodology extending the current industry practice of closed unit tests (i.e. test methods without input parameters). Test methods are generalized by allowing parameters. This serves two purposes. First, parameterized test methods are specifications of the behavior of the methods under test: “they do not only provide exemplary arguments to the methods under test, but ranges of such arguments." [Tillmann 2005]

```
[PexMethod]
public string Capitalize(string value)
{
string result = StringExtensions.Capitalize(value);
return result;
// TODO: add assertions to method …
}
```

The PUT was created in a separate MSTest Project by PEX, as well as a table listing all the actual test cases generated with inputs and outputs:



And the generated test cases are saved in the MSTest project for running by the usual test runner without performing PEX Exploration:



,
[PEX tutorial]

NCover

NCover, which is a command line tool for measuring code coverage, was used to measure code coverage during all test runs.

In this paper we only discuss “block coverage”, as this is the coverage that is supported by the code coverage analysis tool that is built in VS2010.
Block coverage maps roughly to “statement coverage” in [Burnstein 2003]. We are aware that statement coverage is a very weak metric for code coverage. [PEX tutorial]

4.2.2 Test bed

The solution contains several projects:

- RegistationTax - the production code
- Test/PEX projects - MSTest

Every experiment constitutes these steps:

1. Defect seed the code
2. Build the projects
3. Run PEX on relevant test projects to generate new test suite
4. Build the projects
5. Run test suites for all test projects using test runner and measure failures, defects and coverage

4.2.3 Fault detection by test suites

Run the test suites and see which will find the most defects.
We have three variants of test suites:

- EQ+BV, is constant throughout all experiments
- PEX Regression, PEX regression test suite, was run against production code and kept constant in the experiments
- PEX Automatic, is recreated in every experiment

Defect seeding

We will conduct experiments with existing specification and then with added specification. We have found inspiration in the classification of the defects in [Burnstein 2003].

Existing Specs

Algorithmic and processing defect

- We have implemented a spec wrongly.
- We have made a coding error that results in an exception being thrown

Control logic defect, we've forgotten a specification

- Missing branch
- Missing condition from branch

The expected outcome is that our regression suites (EQ+BV and PEX Regression) will catch the introduced defects with regards to the current specification, but PEX Automatic will not.

New Specification

Add specification with "algorithmic and processing defect"

- We've added a specification that throws an exception to simulate defects in the added code

The expected outcome is that our PEX Automatic suite is able to detect the error, but our regression suites will not.

4.3 Test case generation

4.3.1 Outline on Black box test case generation

Black box testing is a common and very powerful test design approach. It involves treating the UUT as a black box – which of course means that the designer of the test cases does not need to have access to the implementation of the UUT. Instead the test cases are being constructed from the specification of the UUT.
The best thing would of course be exhaustive testing, but even for quite simple UUT it is not possible to test using all the combinations of all possible inputs.

This dilemma is what we try to solve: we want a test suite that has a very high probability of revealing defects with as few test cases as possible. The reason that it is an advantage to have as few as possible test cases is that we should also consider that we need to maintain this test code in the rest of the life cycle of the software.

An advantage to Black box test case generation is that it is possible to perform it as soon as the specification is present; that is before the production code is written. As the test cases are created without knowledge to the implementation of the UUT, it is not possible to guarantee that all paths of the code in the UUT are covered. Often though a trained test engineer that follows a systematic approach as well as using his own experience to generate the test cases is likely to get a very high coverage as well as test cases with a high probability of detecting defects. The smart tester [Burnstein 2003] 4.1.

The process of generating test cases can be done by using several techniques all with their own strengths and weaknesses. Examples of these techniques are: equivalence class testing, decision table testing, state transition testing, boundary value testing.

4.3.2 Black box test generation

For our black-box part of the test case generation we decided to perform a thorough equivalence class analysis combined with boundary value analysis. These analyses were then used to create a set of test-cases.

List of conditions:
C1: The price (without VAT) of the car from the dealer
C2: Diesel/Gaz
C3: Mileage (km/liter)
C4: Diesel Particle filter
C5: Seat belt alarms

Condition	Rule/ Heuristic	Invalid EC's	Valid EC's
C1	Range	[i1] price <= 0	[v2] 0 < price < 79.000 [v3] price > 79.000 $[v4]_{bv}$ price = 79.000
C2	Set	[i5] other than [set]	[v6] Gaz [v7] Diesel
C3-1 **GAZ**	Range	[i8] mileage <= 0	[v9] 0 < mileage < 16 [v10] mileage > 16 $[v11]_{bv}$ mileage = 16
C3-2 **DIESEL**	Range		[v12] 0 < mileage < 18 [v13] mileage > 18 $[v14]_{bv}$ 18
C4-1 **GAZ**	Boolean	[i15] Yes	[v16] No
C4-2 **DIESEL**	Boolean		[v17] Yes [v18] No
C5	Range	[i19] alarms < 0 [i20] alarms > 3 $[i21]_{bv}$ alarms = 4	[v22] 0 < alarms <= 3 $[v23]_{bv}$ alarms = 0 $[v24]_{bv}$ alarms = 3

The expected results have been calculated using a test oracle [Burnstein 2003]. The test oracle was a spreadsheet which has been manually tested with examples of calculation given from the website that describes the law [LAW].

Test Case ID	EC combination	Test case	Expected result
TC1	[v2] [v6] [v9] [v16] [v22]	C1=25.000, C2=Gaz, C3=12, C4=No, C5=2	67.662,50
TC2	[v3] [v6] [v10] [v16] [v23]	C1=120.000, C2=Gaz, C3=18, C4=No, C5=0	352.750,00
TC3	[v4] [v6] [v11] [v16] [v24]	C1=79.000, C2=Gaz, C3=16, C4=No, C5=3	216.650,00
TC4	[v2] [v7] [v12] [v15] [v22]	C1=25.000, C2=Diesel, C3=14, C4=Yes, C5=2	64.162,50
TC5	[v3] [v7] [v13] [v17] [v22]	C1=120.000, C2=Diesel, C3=22, C4=Yes, C5=2	340.850,00
TC6	[v4] [v7] [v14] [v17] [v22]	C1=79.000, C2=Diesel, C3=18, C4=Yes, C5=2	213.350,00
TC7	[v4] [v7] [v14] [v18] [v22]	C1=79.000, C2=Diesel, C3=18, C4=No, C5=2	216.850,00
TC8	[i1] [v6] [v9] [v16] [v22]	C1=-25.000, C2=Gaz, C3=12, C4=No, C5=2	Rejected
~~TC9~~	~~[v2] [i5] [v9] [v16] [v20]~~	~~C1=25.000, C2=Petrol, C3=12, C4=No, C5=2~~	~~Rejected~~
TC10	[v2] [v6] [i8] [v16] [v22]	C1=25.000, C2=Gaz, C3=-12, C4=No, C5=2	Rejected
TC11	[v3] [v6] [v11] [i15] [v22]	C1=120.000, C2=Gaz, C3=16, C4=Yes, C5=2	360.350,00
TC12	[v2] [v6] [v9] [v16] [i19]	C1=25.000, C2=Gaz, C3=12, C4=No, C5=-2	Rejected
TC13	[v2] [v6] [v9] [v16] [i20]	C1=25.000, C2=Gaz, C3=12, C4=No, C5=5	67.462,50
TC14	[v2] [v6] [v9] [v16] [i21]	C1=25.000, C2=Gaz, C3=12, C4=No, C5=4	67.462,50

TC9 is cancelled, as it is not possible to pass this illegal argument to the UUT.

4.3.4 Outline on white box testing

White box testing is based on the structure of the code. Statements and branches are analyzed and test cases are developed that exercise these. It's not feasible to exercise all statements and branches of a system, so WB is mainly for smaller parts of the system. Furthermore there has been established different adequacy criteria's for determining when the UUT has been covered. In WB program-based adequacy criteria's are used: statement coverage, decision coverage, condition coverage, decision/statement coverage, multiple-condition coverage and path coverage.

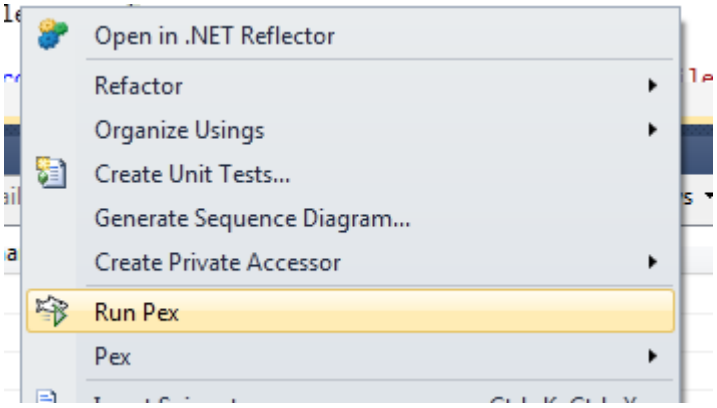
What this implies is that the code is needed to perform the analysis, so compared to BB it's a technique that is used after the UUT has been developed. And furthermore if the UUT changes the analysis will be invalidated. [Burnstein 2003], [Christensen WB]

4.3.5 Automatic white box test generation with PEX

The form of automatic white box testing performed by PEX is dynamic symbolic execution; it runs the code, and tries different branches of the code. (See Appendix 1: Abstract on PEX)

PEX in VS2010

When PEX is installed in Visual Studio there will be additional menus for running PEX. So when you right click inside a method you have the Ability to Run PEX:



When PEX is running it will explore the code by running the code multiple times and generate test cases, the test cases uses a special unit test called a Parameterized Unit test (PUT), to invoke the UUT. The PUT is generated by PEX, but its encouraged to provide additional PUT's and asserts that should hold for all input [PEX Digger].

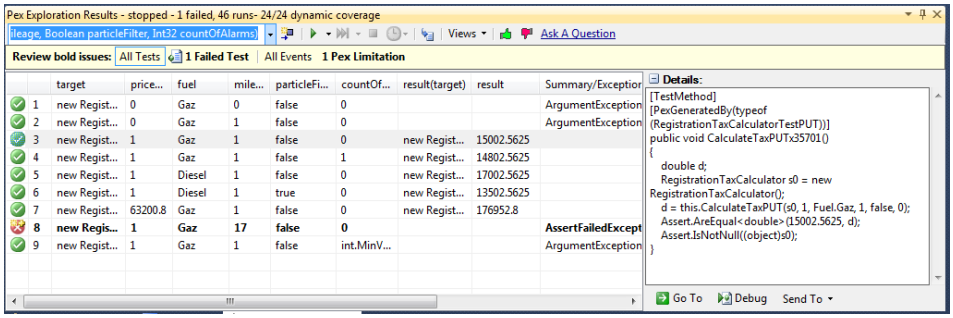
Parameterized Unit test:

```
[PexMethod]
public double CalculateTaxPUT(
    [PexAssumeUnderTest]RegistrationTaxCalculator target,
    double priceBeforeTaxWithoutVAT,
    Fuel fuel,
    double mileage,
    bool particleFilter,
    int countOfAlarms
)
{
    double result = target.CalculateTax
                        (priceBeforeTaxWithoutVAT, fuel, mileage,
particleFilter, countOfAlarms);
}
```

Unit test written against the above PUT:

```
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestPUT))]
[ExpectedException(typeof(ArgumentException))]
public void CalculateTaxPUTThrowsArgumentExceptionx153()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxPUT(s0, 0, Fuel.Gaz, 0, false, 0);
}
```

All tests are shown in a table, and the test case can be seen as a unit test.



The tests can be exported to a separate test project, for re-run, or for adding additional tests and asserts.

4.3.6 What did we learn from the PEX output

How can we use the output from PEX to learn about our implementation? This is not the focus of our hypothesis, but still it shows some of the strengths of PEX.

After implementing our production code, we ran PEX on it:

priceBefor...	fuel	mileage	particleFilter	countOfAl...	result	Error Message
0	Gaz	0	false	0		must be positiveParameter name: mileage
0	Gaz	1	false	0		must be less than 10000000000 and greater ...
1	Gaz	1	false	0	15002.5625	
1	Gaz	17	false	0	-3997.4375	
63200.8	Gaz	1	false	0	176952.8	
1	Diesel	1	false	0	17002.5625	
1	Diesel	1	true	0	13502.5625	
1	Gaz	1	false	int.MinValue		must be positive or zeroParameter name: c...
1	Gaz	1	false	1	14802.5625	
1	Gaz	1	false	5	14402.5625	

What stands out clearly is that we have a negative value somewhere; it's not stated in the specification that we cannot have a negative registration tax, but maybe that is a specification defect? [Burnstein 2003]

Furthermore this table is something we can take with us to the stakeholders and they should be able to relate to it, and tell us if these test cases are valid.

Also what this showed us, was that our EQ+BV analysis was wrong, we had forgotten about the particle filter that only counted on a diesel car. This made us revisit the EQ+BV analysis and repartition, and adjust the test cases.

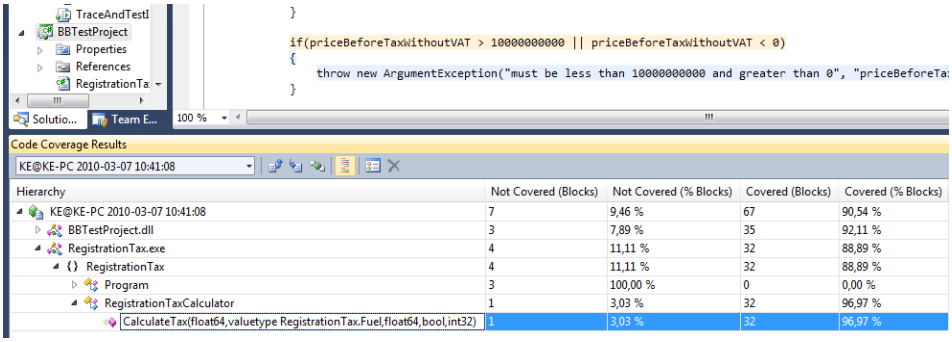
4.4 Coverage results

As outlined in the chapter "How we conduct the experiments" we only perform block coverage using VSTS Coverage tool.

4.4.1 EC coverage result

The EC test suite resulted as expected in a very high coverage. The block coverage is 96.97%.
It can be argued that the coverage should be 100% but the reason that it is not is that the developer that implemented the specification (UUT) has added a condition that was not part of the specification. The condition ("priceBeforeTaxWithoutVAT > 10000000000") can be seen in the code coverage result diagram below. The discovery of this extra condition would result in one of two outcomes. The first is that the developer that implemented the specification (UUT) removes this condition. The other is that the specification gets updated to also contain this requirement; which in turn would mean that the test case engineer should create a test

case that exercises this requirement. We chose the latter approach in our project so we now have 100% code coverage.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
KE@KE-PC 2010-03-07 10:41:08	7	9,46 %	67	90,54 %
BBTestProject.dll	3	7,89 %	35	92,11 %
RegistrationTax.exe	4	11,11 %	32	88,89 %
{ } RegistrationTax	4	11,11 %	32	88,89 %
Program	3	100,00 %	0	0,00 %
RegistrationTaxCalculator	1	3,03 %	32	96,97 %
CalculateTax(float64,valuetype RegistrationTax.Fuel,float64,bool,int32)	1	3,03 %	32	96,97 %

This shows the code coverage result and the part of the code that is not covered.

This way of getting an overview and graphical representation of the code coverage is very handy and immediately makes the test case engineer aware of untested code.

4.4.2 PEX coverage result

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
soren@SOREN-PDC 2010-03-06 22:58:24	3	13,04 %	20	86,96 %
RegistrationTax.exe	3	13,04 %	20	86,96 %
{ } RegistrationTax	3	13,04 %	20	86,96 %
Program	3	100,00 %	0	0,00 %
RegistrationTaxCalculator	0	0,00 %	20	100,00 %
CalculateTax(float64,valuetype Regis...	0	0,00 %	20	100,00 %

PEX reaches a coverage of 100%.

5 Results

We evaluate our experiments in accordance with the metrics defined.

5.1 Maintainability of test case suites

As outlined in the chapter “Maintainability of test code” we evaluate our generated test suites with regards to maintainability:

5.1.1 Maintainability compliance

EQ+BV

Grade: 4.5

The test cases do not completely adhere to the conventions defined by [Meszaros 2007] and [Osherove 2009] 7.3.1 Naming unit tests. This is because we decided that referring back to the test case table (with the test case number) was a better approach. An example of a test case name is:

```
CalculateTax_TC01_returnsNumber()
```

This tells the reader that the method that is being tested is called “CalculateTax”; that we test it under conditions described in the test case table (TC01) and that it is expected to return a number.

PEX

Grade: 1

When looking at an example of the test cases generated by PEX, it is obvious that it does not follow this naming convention.
The test case is named with the PUT under test and suffixed with a number:

```
public void CalculateTaxx202()
```

This makes it impossible to infer what the test case is trying to accomplish. Only the first part is correct – the name of the PUT (which is the same name as the “method under test”).

5.1.2 Analyzability

EQ+BV

Grade: 5

The structure is fine; we do not clearly state what parts of the code belong to which action in AAA, but we could have done so.
The test cases are isolated, they are simple, variable names are clear, we used an external test oracle to verify output and we only verify a single condition pr. Test.

PEX

Grade: 3

Structure: OK, like our manual test cases it does not state which parts belong to which action.
Principles: The tests are isolated, they are simple, variable names are terrible, clearly generated, but we can see them in the table view of PEX, where they provide a meaningful header.

priceBefor...	fuel	mileage	particleFilter	countOfAl...	result	Error Message
0	Gaz	0	false	0		must be positiveParameter name: mileage
0	Gaz	1	false	0		must be less than 10000000000 and greater ...
1	Gaz	1	false	0	15002.5625	
1	Gaz	17	false	0	-3997.4375	
63200.8	Gaz	1	false	0	176952.8	
1	Diesel	1	false	0	17002.5625	
1	Diesel	1	true	0	13502.5625	
1	Gaz	1	false	int.MinValue		must be positive or zeroParameter name: c...
1	Gaz	1	false	1	14802.5625	
1	Gaz	1	false	5	14402.5625	

Output table from PEX

There's no Evident data, if there was a simple way to deduct the result from the input variable we could have provided that as part of the PUT, no verification of the result besides using the production code as test oracle (useful for regression testing) and we have a single condition pr. Test.

5.1.3 Changeability

Evaluation of how the approaches handle changes of different kind in the production code. We mark the test suite as invalid or valid for the different changes:

	Test suites		
Change	EQ+BV	PEX regression	PEX*
Behavioral	Invalid	Invalid	Valid
Refactoring	Valid	Valid	Valid
Interface	Invalid	Invalid	Valid
Grade	1	1	5

* We accept the fact that PEX will use the production code as test oracle.

EQ+BV and PEX regression

Both EQ+BV and PEX regression share common characteristics.
It's a little hard to say that the test suites are invalidated, but they will require refactoring on behavioral and interface changes, we might even have to start over with EQ+BV.
On behavioral changes we need to alter the expected outputs and maybe the inputs to our test cases.
On interface changes, we need to adapt or rewrite the test cases to fit the new interface.
About adapting, we still have the principle of simple test code so we cannot go about making the adaptation to obscure.
During refactoring (behavior and interface stable) they provide a safety net.

PEX

We accept the fact that PEX will use production code as test oracle.
The test suite generated by PEX is just regenerated in every case.
It's worth noting that PEX has the ability to find paths created intentionally or unintentionally during refactoring.

5.2 Experiment results

The table below is an overview of the experiments that we have performed. The following chapters describe the different experiments in greater detail.

		Tests failed	Defects in place	Defects detected	Test suite generation	Code coverage	Test cases	Test case effiency
		Count	Count	Count	Time	Block	Count	
Basis	EQ+BV	4	3	3	4 hours	100%	13	13,0
	PEX	0	3	0	< 1 minute	100%	7	7,0
Algoritmic defect Mileage above limit	EQ+BV	4	1	1		100%	14	14,0
	PEX Regres	6	1	1		100%	10	10,0
	PEX	0	1	1		100%	10	10,0
Coding defect: Missing branch Count of alarms	EQ+BV	2	1	1		100%	14	14,0
	PEX Regres	1	1	1		100%	10	10,0
	PEX	0	1	0		100%	10	10,0
Control logic defect Particle filter	EQ+BV	1	1	1		100%	14	14,0
	PEX Regres	0	1	0		100%	10	10,0
	PEX	0	1	0		100%	10	10,0
Algoritmic defect Throw exception	EQ+BV	2	1	1		100%	14	14,0
	PEX Regres	1	1	1		100%	10	10,0
	PEX	1	1	1		100%	10	10,0
Spec added Mileage over 100	EQ+BV	0	1	0		96%	14	14,5
	PEX Regres	0	1	0		96%	10	10,4
	PEX	1	1	1		100%	11	11,0

Table 1 – The result table of the performed experiments

5.2.1 Basis

The basis in our experiment is the point where the production code is implemented but is untested.

5.2.2 Defects detected

The basis actually contained three defects that all was related to boundary value issues. Our EQ+BV test suite did detect these defects, but the test suite created by PEX did not. This is of course due to the fact that PEX has no knowledge what so ever of the specification.

5.2.3 Time spent on test case generation

This is one metric where we see a large difference between the two approaches. The EQ+BV approach took approximately four hours to complete whereas the PEX approach took less than one minute. This includes creating a parameterized unit test and the creation of a test suite by running “Run Pex Explorations”.

5.2.4 Code coverage

In both cases the coverage is 100%.

5.2.5 Algorithmic defect - Mileage above limit

Simulates the misinterpretation of a specification – This is a coding defect of sub type “Algorithmic and Processing Defects” [Burnstein 2003].

Changed the line from

```
registrationTax -= mileAgeAboveLimit * UNDER_MILEAGE_RATE;
```

To

```
registrationTax += mileAgeAboveLimit * UNDER_MILEAGE_RATE;
```

Defect detection
This defect was detected by both the ”EQ+BV” and ”PEX regression” test suites but not by the ”PEX” test suite.

5.2.6 Coding defect: Missing branch - # alarms

Simulates the absence of a condition from the specification. This is a Coding defect of sub type “Control, Logic and Sequence Defects” [Burnstein 2003]

Removed the lines

```
if (alarmsUsedInCalculation > MAX_COUNT_OF_ALARMS) {
    alarmsUsedInCalculation = MAX_COUNT_OF_ALARMS;
}
```

Defect detection
This defect was detected by both the EQ+BV and PEX regression test suites but not by the PEX test suite.

5.2.7 Control logic defect - Particle filter

Simulates the misinterpretation of a specification. This is a Coding defect of sub type “Control, Logic and Sequence Defects” [Burnstein 2003]

Replaced the line

```
if (fuel == Fuel.Diesel && particleFilter)
```

with the line

```
if (particleFilter)
```

Defect detection
This defect was detected by the EQ+BV test suite but it was not detected by any of the test suites generated by PEX.

5.2.8 Algorithmic defect – Throw exception

Simulates that the production code throws an unexpected exception. This is a crude way of simulating a coding defect like an algorithmic defect (perhaps a division by zero).

Added a line that throws an Exception

```
if (mileAgeAboveLimit > 0) {
    throw new Exception("Defect seeding");
```

Defect detection
This defect was detected by all three test suites.

5.2.9 Spec added – Mileage over 100

This defect is introduced as a coding defect. We have added this defect to the chain of experiments as we regard it to be quite possible that a developer puts guards/pre validation checks in the code that is not part of the specification.

These lines were added to the production code

```
if (mileage > 100) {
    throw new Exception();
}
```

Defect detection
This defect was only detected by the PEX test suite.

6 Conclusion

6.1 Maintainability

We have graded our test suites with regards to maintainability. The EQ+BV test suite are still the best when it comes to maintainability compliance and analyzability.
With regards to changeability it's our opinion that PEX comes out stronger. The PEX regression suite is not less changeable than EQ+BV, but PEX's ability to discover new paths favors in its way.

6.2 Defects detected

Overall the EQ+BV test suite and the PEX Regression test suite did comparably well; in a single experiment however neither of the two PEX test suites found the defect whereas the EQ+BV test suite did.
Also PEX made us realize that we needed to repartition our EQ+BV.
Is also possibly found a specification defect where it returned a negative value, not something PEX flagged as a defect, but something that a human would immediately react to.

6.3 Time spent on test case generation

In this aspect PEX is far superior over the EQ+BV approach. Even though it will take a bit longer for PEX to run on a larger code base so will the time needed to perform the EQ+BV analysis. Thus we conclude that the larger the codebase the greater the advantage of PEX. Also in the light of PEX's ability to adopt changes in the production code, it has a huge advantage on agile projects, where it's likely that specification will evolve over time.

6.4 Code coverage

Code coverage was high for both approaches. It seemed like EQ had better multiple condition coverage, since PEX failed on a single experiment relating to Control logic defect. This fact also shows how weak block coverage is.
PEX was able to get a block coverage of a 100% when introducing a new specification and finding the defect, which neither regression suites did.

6.5 Best practice

Tabular format:
The tabular format that PEX generates is good for processing by a business analyst. If can also assist developers in finding obvious specification/ coding defects.

PEX is good for finding invalid test cases that leads to exceptions, and also to make sure that you have covered the entire UUT. It cannot find invalid test cases based on specifications.
The test case engineer can instead focus the effort on creating “valid” test cases. A “valid” test case is a test case that exercises the UUT without searching for exceptions. These “valid” test cases should of course test the UUT for correctness which PEX cannot do.

Summary of how PEX could be utilized as a tool:

1. Implement code
2. Run PEX during development
3. Adjust code
4. Repeat 2-3
5. Write valid and obvious invalid test cases based on specifications.
6. Let PEX generate a table for processing by Analyst
7. Keep PEX as regression suite.
8. Refactor the code

6.6 Perspective on PEX

6.6.1 PEX cannot test for correctness

As it is probably obvious to everyone we still think it is appropriate to point out that PEX is not meant to test for correctness. It can only use the production code as a test oracle.

6.6.2 Legacy code

Another use case where PEX has the potential to save us a lot of time is where a codebase of legacy code needs to be refactored.
In this case PEX can be used to create a test suite that captures the current behavior of the system. This test suite can be used as a regression test suite so that the behavior of the system can be preserved throughout the refactoring.
We found a very interesting article about exactly this in [MSDN Sachdeva 2009]

6.7 Hypothesis holds?

Yes, we think that we have proven our hypothesis to be true. It is possible to utilize a tool like PEX to assist in creating test cases and thereby reduce the total time spent on creating test cases.

7 Related work

We acknowledge that we have set out to do much the same as the paper: “On the effectiveness of manual and automatic unit test generation” [Effectiveness 2008]. However our motivations is more concerned with evaluating a new tool (PEX) and thereby examine the possibility to reduce time spent on testing.
It is our opinion that this domain should be further explored and this is our contribution to this.

8 Appendix 1: Abstract on PEX

8.1 Outline how PEX works

The way that PEX works is by dynamic symbolic execution. It will track variables as symbolic expressions, the statement:

```
var y = 3;
y = y*3;
```

Will result in a symbolic value s, that initially will contain 3 and then later y is updated to container y*3. Every time a conditional is encountered containing symbolic values a path constraint is setup containing the symbolic expression, i.e..

```
if( y > 3)
```

then the path constraint will be s*3 > 3

In that way a constraint resolver can be used to calculate the intial values to let the program take the different paths. PEX also uses concrete execution to let the values guide the path, and then uses the path constraints to resolve values for hitting other paths. [Symbolic Daniel et. al]

In that way the symbolic execution is used to create test case by alternating the Path criteria.

9 Appendix 2: Source code

9.1 UUT

```
    public class RegistrationTaxCalculator {
        public double CalculateTax(double priceBeforeTaxWithoutVAT, Fuel fuel,
double mileage, bool particleFilter, int countOfAlarms) {
            const int BASE_AMOUNT = 79000;
            const double VAT = 0.25;
            const double MILEAGE_LIMIT_GAZ = 16;
            const double MILEAGE_LIMIT_DIESEL = 18;
            const int OVER_MILEAGE_RATE = -4000;
            const int UNDER_MILEAGE_RATE = 1000;
            const int COUNT_OF_ALARMS_RATE = -200;
            const int MAX_COUNT_OF_ALARMS = 3;
            const int PARTICLE_FILTER_RATE = -3500;
            const double TAXRATE_BELOW_BASE_AMOUNT = 1.05;
            const double TAXRATE_ABOVE_BASE_AMOUNT = 1.8;

            if (mileage < 0) {
                throw new ArgumentException("must be positive", "mileage");
            }

            if (priceBeforeTaxWithoutVAT > 10000000000 ||
priceBeforeTaxWithoutVAT < 0) {
                throw new ArgumentException("must be less than 10000000000 and
greater than 0", "priceBeforeTaxWithoutVAT");
            }

            if (countOfAlarms < 0) {
                throw new ArgumentException("must be positive or zero",
"countOfAlarms");
            }

            var priceWithVAT = priceBeforeTaxWithoutVAT * (1 + VAT);
            var registrationTax = Math.Min(priceWithVAT, BASE_AMOUNT) *
TAXRATE_BELOW_BASE_AMOUNT;
            if (priceWithVAT > BASE_AMOUNT) {
                registrationTax += (priceWithVAT - BASE_AMOUNT) *
TAXRATE_ABOVE_BASE_AMOUNT;
            }

            var mileageLimit = fuel == Fuel.Diesel ? MILEAGE_LIMIT_DIESEL :
MILEAGE_LIMIT_GAZ;
            var mileAgeAboveLimit = mileage - mileageLimit;
            var mileageRate = mileAgeAboveLimit > 0 ? OVER_MILEAGE_RATE :
UNDER_MILEAGE_RATE;
            registrationTax += Math.Abs(mileAgeAboveLimit) * mileageRate;

            if (countOfAlarms > 0) {
                int alarmsUsedInCalculation = countOfAlarms;
                if (alarmsUsedInCalculation > MAX_COUNT_OF_ALARMS) {
                    alarmsUsedInCalculation = MAX_COUNT_OF_ALARMS;
                }
                registrationTax += alarmsUsedInCalculation *
COUNT_OF_ALARMS_RATE;
            }

            if (fuel == Fuel.Diesel && particleFilter) {
                registrationTax += PARTICLE_FILTER_RATE;
            }

            return registrationTax + priceWithVAT;
        }
    }
}
```

9.2 BB test case suite

```
    [TestClass()]
    public class RegistrationTaxCalculatorTest {
        private RegistrationTaxCalculator uut;


        [TestInitialize()]
        public void MyTestInitialize() {
            uut = new RegistrationTaxCalculator();
        }

        [TestMethod()]
        public void CalculateTax_TC01_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 25000;
            double mileage = 12;
            bool particleFilter = false;
            int countOfAlarms = 2;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz,
mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(67662.50, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC02_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 120000;
            double mileage = 18;
            bool particleFilter = false;
            int countOfAlarms = 0;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz,
mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(352750, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC03_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 79000;
            double mileage = 16;
            bool particleFilter = false;
            int countOfAlarms = 3;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz,
mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(216650, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC04_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 25000;
            double mileage = 14;
            bool particleFilter = true;
            int countOfAlarms = 2;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT,
Fuel.Diesel, mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(64162.50, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC05_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 120000;
            double mileage = 22;
            bool particleFilter = true;
            int countOfAlarms = 2;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT,
Fuel.Diesel, mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(340850, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC06_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 79000;
            double mileage = 18;
            bool particleFilter = true;
            int countOfAlarms = 2;
```

```
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT,
Fuel.Diesel, mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(213350, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC07_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 79000;
            double mileage = 18;
            bool particleFilter = false;
            int countOfAlarms = 2;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT,
Fuel.Diesel, mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(216850, actual);
        }

        [TestMethod()]
        [ExpectedException(typeof(ArgumentException))]
        public void CalculateTax_TC08_rejected() {
            double priceBeforeTaxWithoutVAT = -25000;
            double mileage = 12;
            bool particleFilter = false;
            int countOfAlarms = 2;
            uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz, mileage,
particleFilter, countOfAlarms);
        }

        [TestMethod()]
        [ExpectedException(typeof(ArgumentException))]
        public void CalculateTax_TC08_1_rejected() {
            double priceBeforeTaxWithoutVAT = 10000000001;
            double mileage = 12;
            bool particleFilter = false;
            int countOfAlarms = 2;
            uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz, mileage,
particleFilter, countOfAlarms);
        }

        [TestMethod()]
        [ExpectedException(typeof(ArgumentException))]
        public void CalculateTax_TC10_rejected() {
            double priceBeforeTaxWithoutVAT = 25000;
            double mileage = -12;
            bool particleFilter = false;
            int countOfAlarms = 2;
            uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz, mileage,
particleFilter, countOfAlarms);
        }

        [TestMethod()]
        public void CalculateTax_TC11_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 120000;
            double mileage = 16;
            bool particleFilter = true;
            int countOfAlarms = 2;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz,
mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(360350, actual);
        }

        [TestMethod()]
        [ExpectedException(typeof(ArgumentException))]
        public void CalculateTax_TC12_rejected() {
            double priceBeforeTaxWithoutVAT = 25000;
            double mileage = 12;
            bool particleFilter = false;
            int countOfAlarms = -2;
            uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz, mileage,
particleFilter, countOfAlarms);
        }

        [TestMethod()]
        public void CalculateTax_TC12_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 25000;
```

```
            double mileage = 12;
            bool particleFilter = false;
            int countOfAlarms = 5;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz,
mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(67462.50, actual);
        }

        [TestMethod()]
        public void CalculateTax_TC13_returnsNumber() {
            double priceBeforeTaxWithoutVAT = 25000;
            double mileage = 12;
            bool particleFilter = false;
            int countOfAlarms = 4;
            double actual = uut.CalculateTax(priceBeforeTaxWithoutVAT, Fuel.Gaz,
mileage, particleFilter, countOfAlarms);
            Assert.AreEqual(67462.50, actual);
        }
    }
}
```

9.3 PEX test suite

This is the PUT

```
/// <summary>This class contains parameterized unit tests for
RegistrationTaxCalculator</summary>
[PexClass(typeof(RegistrationTaxCalculator))]
[PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
[PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException),
AcceptExceptionSubtypes = true)]
[TestClass]
public partial class RegistrationTaxCalculatorTestRegression
{
    /// <summary>Test stub for CalculateTax(Double, Fuel, Double, Boolean,
Int32)</summary>
    [PexMethod]
    public double CalculateTaxRegression(
        [PexAssumeUnderTest]RegistrationTaxCalculator target,
        double priceBeforeTaxWithoutVAT,
        Fuel fuel,
        double mileage,
        bool particleFilter,
        int countOfAlarms
    )
    {
        double result = target.CalculateTax
                            (priceBeforeTaxWithoutVAT, fuel, mileage,
particleFilter, countOfAlarms);
        return result;
        // TODO: add assertions to method
RegistrationTaxCalculatorTestRegression.CalculateTaxRegression(RegistrationTaxCa
lculator, Double, Fuel, Double, Boolean, Int32)
    }
}
```

The test cases generated by PEX:

```
    public partial class RegistrationTaxCalculatorTestRegression {
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
[ExpectedException(typeof(ArgumentException))]
public void CalculateTaxRegressionThrowsArgumentExceptionx153()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 0, Fuel.Gaz, 0, false, 0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
[ExpectedException(typeof(ArgumentException))]
public void CalculateTaxRegressionThrowsArgumentExceptionx241()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 0, Fuel.Gaz, 1, false, 0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx357()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 1, Fuel.Gaz, 1, false, 0);
    Assert.AreEqual<double>(15002.5625, d);
    Assert.IsNotNull((object)s0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx913()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
```

```
    d = this.CalculateTaxRegression(s0, 1, Fuel.Gaz, 17, false, 0);
    Assert.AreEqual<double>(-3997.4375, d);
    Assert.IsNotNull((object)s0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx508()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 63200.8, Fuel.Gaz, 1, false, 0);
    Assert.AreEqual<double>(176952.8, d);
    Assert.IsNotNull((object)s0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx217()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 1, Fuel.Diesel, 1, false, 0);
    Assert.AreEqual<double>(17002.5625, d);
    Assert.IsNotNull((object)s0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx348()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 1, Fuel.Diesel, 1, true, 0);
    Assert.AreEqual<double>(13502.5625, d);
    Assert.IsNotNull((object)s0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
[ExpectedException(typeof(ArgumentException))]
public void CalculateTaxRegressionThrowsArgumentExceptionx887()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 1, Fuel.Gaz, 1, false, int.MinValue);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx144()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 1, Fuel.Gaz, 1, false, 1);
    Assert.AreEqual<double>(14802.5625, d);
    Assert.IsNotNull((object)s0);
}
[TestMethod]
[PexGeneratedBy(typeof(RegistrationTaxCalculatorTestRegression))]
public void CalculateTaxRegressionx190()
{
    double d;
    RegistrationTaxCalculator s0 = new RegistrationTaxCalculator();
    d = this.CalculateTaxRegression(s0, 1, Fuel.Gaz, 1, false, 5);
    Assert.AreEqual<double>(14402.5625, d);
    Assert.IsNotNull((object)s0);
}
    }
}
```

10 Appendix 3: Test suite result tables

This appendix lists all the test result tables for each of the test suites ordered by defects.

10.1 Basis

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None]

Test run completed Results: 14/14 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTax_TC01_returnsNumber		
Passed	CalculateTax_TC02_returnsNumber		
Passed	CalculateTax_TC03_returnsNumber		
Passed	CalculateTax_TC04_returnsNumber		
Passed	CalculateTax_TC05_returnsNumber		
Passed	CalculateTax_TC06_returnsNumber		
Passed	CalculateTax_TC07_returnsNumber		
Passed	CalculateTax_TC08_1_rejected		
Passed	CalculateTax_TC08_rejected		
Passed	CalculateTax_TC10_rejected		
Passed	CalculateTax_TC11_returnsNumber		
Passed	CalculateTax_TC12_rejected		
Passed	CalculateTax_TC13_returnsNumber		
Passed	CalculateTax_TC14_returnsNumber		

Table 2 - "EQ+BV" - Basis

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None]

Test run completed Results: 10/10 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTaxThrowsArgumentExceptionx24		
Passed	CalculateTaxThrowsArgumentExceptionx252		
Passed	CalculateTaxThrowsArgumentExceptionx643		
Passed	CalculateTaxx144		
Passed	CalculateTaxx190		
Passed	CalculateTaxx217		
Passed	CalculateTaxx348		
Passed	CalculateTaxx357		
Passed	CalculateTaxx508		
Passed	CalculateTaxx913		

Table 3 - "PEX" - Basis

10.2 Algorithmic defect - Mileage above limit

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None] | [All Columi | <Typ

Test run failed Results: 10/14 passed; Item(s) checked: 4

Result	Test Name	Project	Error Message
Failed	CalculateTax_TC01_returnsNumber		Assert.AreEqual failed. Expected:<67662,5>. Actual:<59662,5>.
Passed	CalculateTax_TC02_returnsNumber		
Passed	CalculateTax_TC03_returnsNumber		
Failed	CalculateTax_TC04_returnsNumber		Assert.AreEqual failed. Expected:<64162,5>. Actual:<56162,5>.
Passed	CalculateTax_TC05_returnsNumber		
Passed	CalculateTax_TC06_returnsNumber		
Passed	CalculateTax_TC07_returnsNumber		
Passed	CalculateTax_TC08_1_rejected		
Passed	CalculateTax_TC08_rejected		
Passed	CalculateTax_TC10_rejected		
Passed	CalculateTax_TC11_returnsNumber		
Passed	CalculateTax_TC12_rejected		
Failed	CalculateTax_TC13_returnsNumber		Assert.AreEqual failed. Expected:<67462,5>. Actual:<59462,5>.
Failed	CalculateTax_TC14_returnsNumber		Assert.AreEqual failed. Expected:<67462,5>. Actual:<59462,5>.

Table 4 – "EQ+BV" - Algorithmic defect - Mileage above limit

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None] | [All Columr | <Type keywo

Test run failed Results: 4/10 passed; Item(s) checked: 6

Result	Test Name	Project	Error Message
Passed	CalculateTaxRegressionThrowsArgumentExceptionx1		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx2		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx8		
Failed	CalculateTaxRegressionx144		Assert.AreEqual failed. Expected:<14802,5625>. Actual:<-15197,4375>.
Failed	CalculateTaxRegressionx190		Assert.AreEqual failed. Expected:<14402,5625>. Actual:<-15597,4375>.
Failed	CalculateTaxRegressionx217		Assert.AreEqual failed. Expected:<17002,5625>. Actual:<-16997,4375>.
Failed	CalculateTaxRegressionx348		Assert.AreEqual failed. Expected:<13502,5625>. Actual:<-20497,4375>.
Failed	CalculateTaxRegressionx357		Assert.AreEqual failed. Expected:<15002,5625>. Actual:<-14997,4375>.
Failed	CalculateTaxRegressionx508		Assert.AreEqual failed. Expected:<176952,8>. Actual:<146952,8>.
Passed	CalculateTaxRegressionx913		

Table 5 - "PEX Regression" - Algorithmic defect - Mileage above limit

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [Nor

Test run completed Results: 10/10 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTaxThrowsArgumentExceptionx24		
Passed	CalculateTaxThrowsArgumentExceptionx252		
Passed	CalculateTaxThrowsArgumentExceptionx643		
Passed	CalculateTaxx464		
Passed	CalculateTaxx542		
Passed	CalculateTaxx688		
Passed	CalculateTaxx706		
Passed	CalculateTaxx79		
Passed	CalculateTaxx882		
Passed	CalculateTaxx913		

Table 6 - "PEX" - Algorithmic defect - Mileage above limit

10.3 Coding defect: Missing branch - # alarms

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None]

Test run failed Results: 12/14 passed; Item(s) checked: 2

Result	Test Name	Project	Error Message
Passed	CalculateTax_TC01_returnsNumber		
Passed	CalculateTax_TC02_returnsNumber		
Passed	CalculateTax_TC03_returnsNumber		
Passed	CalculateTax_TC04_returnsNumber		
Passed	CalculateTax_TC05_returnsNumber		
Passed	CalculateTax_TC06_returnsNumber		
Passed	CalculateTax_TC07_returnsNumber		
Passed	CalculateTax_TC08_1_rejected		
Passed	CalculateTax_TC08_rejected		
Passed	CalculateTax_TC10_rejected		
Passed	CalculateTax_TC11_returnsNumber		
Passed	CalculateTax_TC12_rejected		
Failed	CalculateTax_TC13_returnsNumber		Assert.AreEqual failed. Expected:<67462,5>. Actual:<67062,5>.
Failed	CalculateTax_TC14_returnsNumber		Assert.AreEqual failed. Expected:<67462,5>. Actual:<67262,5>.

Table 7 - "EQ+BV" - Coding defect: Missing branch - # alarms

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None] | [All Columr | <Type keyw

Test run failed Results: 9/10 passed; Item(s) checked: 1

Result	Test Name	Project	Error Message
Passed	CalculateTaxRegressionThrowsArgumentExceptionx1		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx2		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx8		
Passed	CalculateTaxRegressionx144		
Failed	CalculateTaxRegressionx190		Assert.AreEqual failed. Expected:<14402,5625>. Actual:<14002,5625>.
Passed	CalculateTaxRegressionx217		
Passed	CalculateTaxRegressionx348		
Passed	CalculateTaxRegressionx357		
Passed	CalculateTaxRegressionx508		
Passed	CalculateTaxRegressionx913		

Table 8 - "PEX Regression" - Coding defect: Missing branch - # alarms

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [Nor

Test run completed Results: 9/9 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTaxThrowsArgumentExceptionx172		
Passed	CalculateTaxThrowsArgumentExceptionx302		
Passed	CalculateTaxThrowsArgumentExceptionx457		
Passed	CalculateTaxx144		
Passed	CalculateTaxx217		
Passed	CalculateTaxx348		
Passed	CalculateTaxx357		
Passed	CalculateTaxx508		
Passed	CalculateTaxx913		

Table 9 - "PEX" - Coding defect: Missing branch - # alarms

10.4 Control logic defect - Particle filter

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [None] | [All Columr | <Ty

Test run failed Results: 13/14 passed; Item(s) checked: 1

Result	Test Name	Project	Error Message
Passed	CalculateTax_TC01_returnsNumber		
Passed	CalculateTax_TC02_returnsNumber		
Passed	CalculateTax_TC03_returnsNumber		
Passed	CalculateTax_TC04_returnsNumber		
Passed	CalculateTax_TC05_returnsNumber		
Passed	CalculateTax_TC06_returnsNumber		
Passed	CalculateTax_TC07_returnsNumber		
Passed	CalculateTax_TC08_1_rejected		
Passed	CalculateTax_TC08_rejected		
Passed	CalculateTax_TC10_rejected		
Failed	CalculateTax_TC11_returnsNumber		Assert.AreEqual failed. Expected:<360350>. Actual:<356850>.
Passed	CalculateTax_TC12_rejected		
Passed	CalculateTax_TC13_returnsNumber		
Passed	CalculateTax_TC14_returnsNumber		

Table 10 - "EQ+BV" - Control logic defect - Particle filter

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [Non

Test run completed Results: 10/10 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTaxRegressionThrowsArgumentExceptionx1		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx2		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx8		
Passed	CalculateTaxRegressionx144		
Passed	CalculateTaxRegressionx190		
Passed	CalculateTaxRegressionx217		
Passed	CalculateTaxRegressionx348		
Passed	CalculateTaxRegressionx357		
Passed	CalculateTaxRegressionx508		
Passed	CalculateTaxRegressionx913		

Table 11 - "PEX Regression" - Control logic defect - Particle filter

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [Non

Test run completed Results: 10/10 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTaxThrowsArgumentExceptionx556		
Passed	CalculateTaxThrowsArgumentExceptionx761		
Passed	CalculateTaxThrowsArgumentExceptionx930		
Passed	CalculateTaxx144		
Passed	CalculateTaxx190		
Passed	CalculateTaxx217		
Passed	CalculateTaxx357		
Passed	CalculateTaxx404		
Passed	CalculateTaxx508		
Passed	CalculateTaxx913		

Table 12 - "PEX" - Control logic defect - Particle filter

10.5 Spec. added – Mileage over 100

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [Non

Test run completed Results: 14/14 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTax_TC01_returnsNumber		
Passed	CalculateTax_TC02_returnsNumber		
Passed	CalculateTax_TC03_returnsNumber		
Passed	CalculateTax_TC04_returnsNumber		
Passed	CalculateTax_TC05_returnsNumber		
Passed	CalculateTax_TC06_returnsNumber		
Passed	CalculateTax_TC07_returnsNumber		
Passed	CalculateTax_TC08_1_rejected		
Passed	CalculateTax_TC08_rejected		
Passed	CalculateTax_TC10_rejected		
Passed	CalculateTax_TC11_returnsNumber		
Passed	CalculateTax_TC12_rejected		
Passed	CalculateTax_TC13_returnsNumber		
Passed	CalculateTax_TC14_returnsNumber		

Table 13 - "EQ+BV" - Spec. added – Mileage over 100

Test Results

soren@SOREN-PDC 2010-03-13 2 | Run | Debug | Group By: [Non

Test run completed Results: 10/10 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	CalculateTaxRegressionThrowsArgumentExceptionx1		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx2		
Passed	CalculateTaxRegressionThrowsArgumentExceptionx8		
Passed	CalculateTaxRegressionx144		
Passed	CalculateTaxRegressionx190		
Passed	CalculateTaxRegressionx217		
Passed	CalculateTaxRegressionx348		
Passed	CalculateTaxRegressionx357		
Passed	CalculateTaxRegressionx508		
Passed	CalculateTaxRegressionx913		

Table 14 - "PEX Regression" - Spec. added – Mileage over 100

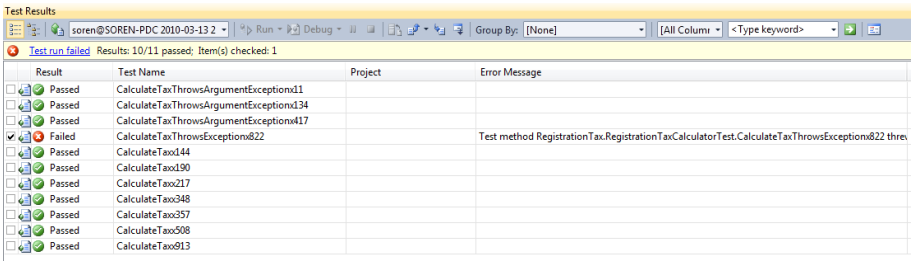


Table 15 - "PEX" - Spec. added – Mileage over 100

11 References

[Pfaller 2008]	Multi-Dimensional Measures for Test Case Quality - Christian Pfaller et. Al - 2008
[LAW]	The "Registreringsafgiftsloven": http://www.skm.dk/tal_statistik/satser_og_beloeb/228.html is a law that is used to determine how much money it costs to register a vehicle in Denmark.
[Calculator]	The test oracle used to calculate the expected result for the test cases and to verify the correctness of the test case results.
[Burnstein]	Practical Software Testing - Ilene Burnstein, Springer-Verlag - 2003
[RSA]	RSA Compilation AU Computer Science Department, 2009
[ISO SW Quality]	Software engineering – Product quality – Part 1: Quality model
[Effectiveness]	On the effectiveness of manual and automatic unit test generation - Alberto Bacchelli et. Al.
[Osherove]	The art of unittesting - Roy Osherove – 2009
[Meszaros]	xUnit Test Patterns – Gerard Meszaros – 2007
[Tilmann 2005]	http://research.microsoft.com/pubs/77417/p241-tillmann.pdf or http://portal.acm.org/citation.cfm?doid=1081706.1081745
[PEX tutorial]	http://research.microsoft.com/en-us/projects/pex/pextutorial.pdf
[FRSE 2010]	[Reliable and Flexible Software Explained – Henrik Bærbak Christensen – 2010]
[Daniel et. al]	Daniel, B., Gvero, T., & Marinov, D. (n.d.). On Test Repair using Symbolic Execution http://mir.cs.illinois.edu/reassert/pubs/symreassert.pdf.

[Christensen WB]	http://www.cs.au.dk/rsa/notes/whitebox.ppt
[PEX Digger]	http://research.microsoft.com/en-us/projects/pex/digger.pdf
[MSDN Sachdeva 2009]	http://msdn.microsoft.com/en-us/magazine/ee819140.aspx