



Chapter

34

Systematic Testing

Learning Objectives

The test-driven development approach puts much emphasis on automated tests and several TDD principles emphasize the importance of quality test cases, such as *Evident Tests* and *Evident Data*. However, tests can only demonstrate the presence of defects, not that no defects remain in the production code. The learning objective of this chapter is to present a few systematic testing techniques that increase the chance of finding defects while keeping the number of test cases low. Thus systematic testing is not an alternative to TDD but rather a different aspect of testing: TDD is focused on the process of building reliable software fast while systematic testing is focused on increasing the ability of test cases to expose defects. As such they complement each other.

34.1 Terminology

As defined in Chapter 2 a failure is the situation in which the system's behavior deviates from the expected, and is caused by a defect in the production code. Thus, if I can reduce the number of defects in my software, it will exhibit fewer failures and thus increase the software's reliability.

Exercise 34.1: Actually, this statement is not always true. Find situations where A) a removed defect in the code does not alter the resulting system's reliability, and B) removing defect 1 gives a high increase in reliability while removing defect 2 gives a low increase in reliability.

It should therefore be obvious that techniques that increase the likelihood of finding a defect are important and interesting in software engineering. These techniques are called *systematic testing* techniques.

Definition: Systematic testing

Systematic testing is a planned and systematic process with the explicit goal of finding defects in some well-defined part of the system.

Note by this definition testing is a *destructive* process in contrast to almost all other processes in software development which are *constructive*. In testing, *your criteria of success is to prove that the system does not work!* This is perhaps one of the reasons why it is so hard—developers are naturally reluctant to prove they have done a poor job!

Research and practice have evolved a large number of techniques over the years. Generally, these techniques are classified into two major classes.

Definition: Black-box testing

The *unit under test* (UUT) is treated as a black box. The only knowledge we have to guide our testing effort is the specification of the UUT and a general knowledge of common programming techniques, algorithmic constructs, and common mistakes made by programmers.

Definition: White-box testing

The full implementation of the UUT is known, so the actual code can be inspected in order to generate test cases.


I will only discuss black-box testing techniques in this book. You will find some references to books that have a comprehensive overview of techniques in the summary section at the end of the chapter.

Generally, the systematic techniques are quite demanding and thus costly in terms of effort you have to invest. You should always balance the invested effort with the expected increase of reliability. As an example, consider a modern web browser. A web browser has numerous options you can set, however, the vast majority of users never change these options from their default setting. From a return-on-investment point of view it therefore makes sense to test the browser with the default options set much more thoroughly than when the options are set to specialized values.

The complexity of the unit under test is also important to consider when picking your testing strategy. I find the following classification of testing approaches appropriate based upon the complexity of the UUT.

- *No testing.* Many methods are so small that they are not worth testing. Examples are accessor methods that can be assumed simply to return the value of some instance variable. The testing code for such a method will become longer than the production code and thus increase the probability of tests failing due to defects in the test code rather than the production code.
- *Explorative testing.* Explorative tests are tests you make based on experience and “gut feeling” but you do not follow any rigid method. The explorative tests are well suited for medium complex methods and are characterized by their low cost. They are quite efficient as you gain experience as a tester and TDD developer. The test-driven development process basically uses an explorative test strategy as a fast development cycle is considered very important.

- *Systematic testing.* Here you follow a rigid method for generating test cases in order to increase the probability of finding defects. Systematic testing is costly as quite a lot of effort is invested in careful analysis of the problem. Thus this technique is best used for highly complex methods where the investment is worthwhile; or systems where reliability is of utmost importance such as machinery that may pose a safety risk for humans or the environment if they fail.

 Review the chapter on test-driven development using the classification above to see if you can find examples of no testing, explorative testing, and systematic testing.

Below, I will present two central black box testing techniques: *equivalence class partitioning* and *boundary value analysis*.

34.2 Equivalence Class Partitioning

The equivalence class partitioning technique relies on the fact that many input values are treated alike by our programs. To motivate this, let us consider a very simple example, the method `int Math.abs(int x)` in the Java system libraries. This simple method calculates the absolute value of an integer: *If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.* One test case table for this function could look like this:

Unit under test: Math.abs	
Input	Expected output
x = 37	37
x = 38	38
x = 39	39
x = 40	40
x = 41	41

The question is if I can be sure that the function is reliably implemented based upon these five test cases and if these particular five test cases are the best test cases to pick. The answer to both questions is *no*. The specification of `Math.abs` states that *If the argument is not negative, the argument is returned* and thus all the input values above are treated alike. Any competent programmer will write an implementation in which all the above input values are handled by the same code fragment. Thus if the `x=37` test case passes then so will `x=38` etc. I do not find more defects by adding more test cases for positive `x` values.

On the other hand, the test case table above has no test cases for negative `x`. Thus the code fragment to handle negative arguments is simply not exercised and the test cases above will not catch a faulty implementation like

```
public static int abs(int x) { return x; }
```

This insight is the core of the equivalence partitioning technique: find a single input value that represents a large set of values. If the test case $x=37$ exposes a defect in the implementation then $x=41$ will probably expose the same defect. You say that 37 is a **representative** for the class of all positive integers when testing `Math.abs`. Likewise I can choose $x=-42$ to represent all negative values. Thus the input space becomes partitioned into *equivalence classes*.

Definition: Equivalence class (EC)

A subset of all possible inputs to the UUT that has the property that if one element in the subset demonstrates a defect during testing, then we assume that all other elements in the subset will demonstrate the *same* defect.

Thus I have reduced the vast input space of `abs` into just two ECs and just two test cases are needed. These two test cases have a high probability of finding defects as they represent all aspects of the absolute value specification.

In the `Math.abs` examples, all possible input values are valid but often this is not the case. As an example, consider `Math.sqrt(double x)` (square root of a number) that is undefined for negative x . Therefore a distinction is made between **valid equivalence classes** whose elements will be processed normally, and **invalid equivalence classes** whose elements define special processing, like throwing an exception, returning an undefined value, or otherwise result in abnormal processing. I find that the terms “valid” and “invalid” are a bit unfortunate as it is sometimes a good idea to classify ECs as invalid even though they strictly speaking are valid to the method. The reason I never-the-less stick to this terminology is because it is widespread in the literature. My rule of thumb is that if elements from the EC will typically be processed early in the algorithm by simple switches and lead to the method *bailing out* then I classify it as invalid. You will see examples of this rule of thumb applied in examples later, as well as see why it is important.

When partitioning the input space into ECs, two properties must be fulfilled in order for the partitioning to be sound:

- **Coverage:** Every possible input element belongs to at least one of the equivalence classes.
- **Representation:** If a defect is demonstrated on a particular member of an equivalence class, the same defect is assumed to be demonstrated by any other member of the class.

 Argue that the two properties are fulfilled for the `Math.abs` example.

A good and handy way to document a set of ECs is by an **equivalence class table**. I will use the format below, here shown on the `abs` example.

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[a1]$ $x \leq 0[a2]$

Copyrighted Material. Do Not Distribute.

The first column describes the condition that has led to partitioning, as outlined in the next section, and the next two columns describe the invalid and valid ECs. I generally prefer inserting the specification of the EC directly into the table, using a semi-mathematical set notation, and give each EC a label, like [a1] and [a2] above, so they are easy to refer to later. A proper mathematical formulation of EC [a1] would be $\{x \in \mathbb{Z} | x > 0\}$ but to conserve space I will usually write it as “ $x > 0$ ” or even just “ > 0 ”. You can also choose to write out the EC in natural language in a separate list and just use the labels as cross references, like:

Condition	Invalid ECs	Valid ECs
absolute value of x	–	[a1] [a2]

where

- [a1] are positive integers
- [a2] are negative integers.

34.2.1 Finding the Equivalence Classes

Unfortunately finding a good set of ECs is often a difficult process that requires skills and experience. Below I will present a set of guidelines and heuristics, adapted and extended from Myers (1979), that are helpful but remember that there are no hard and fast rules. The best way forward is an iterative process where you refine the ECs and test cases as you gain insight into the problem: *take small steps* is a valuable principle in systematic testing as well.

Conditions

A good source for partitioning is to look for *conditions* in the specifications of the unit under test. These conditions are often associated with the input values to the unit (as was the case for the absolute value method) but sometimes also on its output (the formatting method in Section 34.2.6 is an example).

Given a condition, you can derive a first set of ECs following the guidelines below:

- **Range:** *If a condition is specified as a range of values*, select one valid EC that covers the allowed range, and two invalid ECs, one above and one below the end of the range.
- **Set:** *If a condition is specified as a set of values* then define an EC for each value in the set and one EC containing all elements outside the set.
- **Boolean:** *If a condition is specified as a “must be” condition* then define one EC for the condition being true and one EC for the condition being false.

As an example of the set guideline, consider the pay station whose specification states that it must accept 5, 10, and 25 cent coins. This is the set of valid values and thus four ECs emerge:

Condition	Invalid ECs	Valid ECs
Allowed coins	$\notin \{5, 10, 25\} [a1]$	$\{5\} [a2]; \{10\} [a3]; \{25\} [a4]$

Remember that ECs are mathematical sets and thus set notation often comes in handy. I, however, often write the ECs rather informally but readable (as is the case with the invalid EC above) as ECs are means to generate test cases more than an end in themselves.

An example of the boolean guideline, consider a method to recognize a properly formatted programming language identifier that is required to start with a letter. This is clearly a “must be” condition leading to two ECs.

Condition	Invalid ECs	Valid ECs
Initial character of identifier	non-letter $[a1]$	letter $[a2]$

Finally, for the range guideline, consider a method to test if a position on a standard chess board numbering columns a–h and rows 1–8 is valid. It is of course not valid to specify positions outside the board, leading to these six ECs:

Condition	Invalid ECs	Valid ECs
Column	$< 'a' [a1]; > 'h' [a2]$	$'a'-'h' [a3]$
Row	$< 1 [b1]; > 8 [b2]$	$1-8 [b3]$

The examples later in the chapter will demonstrate further uses of the guidelines. Note that I construct labels by both enumerating the condition and the partition to ease reading them: $[a1]$ is the first partition of the column condition, while $[b3]$ is the third partition of the row condition. This makes it a bit easier to read them in the extended test case tables that I will introduce shortly.

These guidelines are rooted in how programmers would normally handle the three types of conditions. Ranges are typically coded by conditional statements guarding the ends of the range

```
if ( row < 1 || row > 8 ) { ... }
```

Sets are often handled by membership testing, like

```
if ( coin == 5 || coin == 10 || coin == 25 ) { ... }
// alternative
switch ( coin ) { case 5: case 10: case 25: { ... } }
```

or by putting elements into a data structure and then testing for membership. In any case there are specific and potentially defective code fragments handling each member of the set: either the case or conditionals, or the code that insert members into the data structure. Note that if a set contains a large number of members it may be infeasible to make ECs for each member. However, be sure to include at least elements inside and outside the set.

Computations

Looking for conditions in the specification works fine—unless there are not any. Some methods or functions are computations that calculate output values based on arithmetic operations like addition, multiplication, etc., and they are not handled by the above heuristics. Trying to use them leave you with only a single EC and thereby only one test case with little guidance on picking test input that comprehensively tests the computation.

As an example, consider the following method `linearFunction`

```
// return a*x + b
public double linearFunction(double a, double x, double b) {
    return a*x + b;
}
```

This method has no conditions on ranges, sets, nor boolean conditions. And therefore, I am not helped in finding any ECs for it and may then be lured into having only a single EC

Condition	Invalid ECs	Valid ECs
(none)	-	any a, any x, any b [a1]

As I have no help in picking any representation for any of the values a, x, or b, then a test case table may look like this

ECs covered	Test case	Expected output
[a1]	a=0.0; x=0.0; b=0.0;	0.0

which would never catch a Fake-it implementation like this:

```
// return a*x + b
public double linearFunction(double a, double x, double b) {
    return 0.0;
}
```

Also if I make several test cases where input parameter “a” always happens to be 1.0 then I cannot catch a defect where the programmer forgot to multiply “a”, like

```
// return a*x + b
public double linearFunction(double a, double x, double b) {
    return x + b;
}
```

Similar, if I make several test cases where “b” always happens to be 0.0 then I cannot catch a defect like:

```
// return a*x + b
public double linearFunction(double a, double x, double b) {
    return a*x;
}
```


Thus, there is something about computations that makes the numbers 0.0 and 1.0 less ideal in testing. The insight is to treat *mathematical neutral elements* as distinct ECs.

Therefore the set of heuristics must be extended with the following:

If the specification of the UUT test defines an *arithmetic computation*, then

- **Addition and subtraction:** *If a computation includes addition or subtraction, select one valid EC for the neutral element 0, and one valid EC for all other elements.*
- **Multiplication and division:** *If a computation includes multiplication or division, select one valid EC for the neutral element 1, and one valid EC for all other elements.*

These heuristics must be applied individually to all elements in the computation. As there is one multiplication and one addition in the `linearFunction()`, I end with a EC table like:

Condition	Invalid ECs	Valid ECs
multiplication ($a * x$)	-	$a = 1 [a1]; a \neq 1 [a2]$
	-	$x = 1 [a3]; x \neq 1 [a4]$
addition ($+b$)	-	$b = 0 [b1]; b \neq 0 [b2]$

The rules above are formulated for the arithmetic operations but of course the same argumentation must be applied for other operators with neutral elements, such as set operations, relational algebra, vector spaces, matrices, etc.

34.2.2 Generating the Test Cases

Once the ECs have been established I can generate test cases by picking elements from each EC. For primitive units where the ECs are disjoint this is trivial, consider the absolute value example:

ECs covered	Test case	Expected output
$[a1]$	$x = 201$	+201
$[a2]$	$x = -87$	+87

I have documented the generated test case by an **extended test case table** in which I write the number(s) of the EC(s) that the test case input belongs to in the first column.

More often, however, the ECs are generated from a set of conditions and they therefore overlap. In this case, you have to make test cases by combining the ECs. A simple example is the chess position validation method above in which you cannot only supply e.g. the row parameter to the method without also providing a column parameter. Thus any test case for this unit draws upon elements from two ECs. The test cases could be:

ECs covered	Test case	Expected output
[a1], [b1]	(' ',0)	illegal
[a2], [b1]	('i',-2)	illegal
[a3], [b1]	('e',0)	illegal
[a1], [b2]	(' ',9)	illegal
[a2], [b2]	('j',9)	illegal
[a3], [b2]	('f',12)	illegal
[a1], [b3]	(' ',4)	illegal
[a2], [b3]	('i',5)	illegal
[a3], [b3]	('b',6)	legal

As you imagine, the set of test cases can become very large if there are many ECs for many independent conditions. Essentially you get a combinatorial explosion as each EC must be combined with all ECs developed for all independent conditions. You see it in the above where the column EC [a1] has to be combined with all row ECs [b1], [b2], [b3] to cover all combinations. To limit the number of test cases, Myers (1979) suggested the following heuristics:

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose element only lies in a single invalid ECs.

These guidelines allow me to reduce the number of test cases for the above example somewhat. The first rule concerning valid ECs does not change anything because there is only one valid test case for the [a3], [b3] combination. However, for the invalid test cases there is a change, as most of the test cases in the above table combine invalid EC for both column and row. If only one is allowed to be invalid at the time I get:

ECs covered	Test case	Expected output
[a1], [b3]	(' ',5)	illegal
[a2], [b3]	('j',3)	illegal
[a3], [b1]	('b',0)	illegal
[a3], [b2]	('c',9)	illegal
[a3], [b3]	('b',6)	legal

I have thus reduced the number of test cases from the original nine to now only five. If you have more conditions and more partitions for each condition the reduction in number of test cases is even greater.

There is a strong argumentation for these guidelines. Taking the guideline for letting only one condition be invalid at a time first, it is actually a way to avoid test cases that pass for the wrong reason due to **masking**. To see masking in action, consider an (incomplete) implementation of the chess board position method that checks its parameters one by one:

Listing: chapter/blackbox-test/ChessBoard.java

```
/** Demonstration of masking of defects.
 */
```

Copyrighted Material. Do Not Distribute.

```

public class ChessBoard {
    public boolean valid(char column, int row) {
        if ( column < 'a' ) { return false; }
        if ( row < 0 ) { return false; }
        return true;
    }
}

```

Note that the column check is correct with respect to the low boundary but the row check is not—it should have read “row < 1”. However, the test case (‘ ’, 0) derived from ECs [1], [4] correctly pass as the method indeed returns false to indicate an illegal position. This is an example of masking that leads to the tester making the wrong conclusion: The production code is deemed correct as the test passes but it is indeed defective. The correct column checking code line *masks* for the defect in the row checking code line. By *only* making *one* condition invalid at a time you avoid the masking problem. The test case table generated based on Myers’ heuristics will catch the erroneous row checking line.

The rule to cover as many valid ECs as possible is also driven from facts about the production code. As we must assume that any provided valid input values to a unit under test is used in its algorithms at some point there is no masking problem for valid values.

Finally, for computations I include a rule

For computations, define test cases that only include ECs with non-neutral elements.

The reason is that neutral elements are “invisible” in computation and thus I cannot test whether the operator is present or not if I use a neutral element as test input. That is, for our `linearFunction()` introduced earlier, if I have $a * x$ then if “a” is 1 then we cannot test if the computation is really $a * x$ or just x . Thus the EC with $a = 1$ is not providing any value, so why include it?

And then we use the test case generating heuristic for computations, that is, leave out the ECs for neutral elements

ECs covered	Test case	Expected output
[a2], [a4], [b2]	a=1.3; x=2.7; b=3.3;	6.81

None of the defective implementations of `linearFunction` above would escape detection by this single test case.

34.2.3 The Process

The process of equivalence partitioning is roughly the same every time:

1. Review the requirements for the UUT and identify *conditions* and use the heuristics to find ECs for each condition. ECs are best written down in an *equivalence class table*.

2. Review the produced ECs and consider carefully the representation property of elements in each EC. If you question if particular elements are really representative then repartition the EC.
3. Review to verify that the coverage property is fulfilled.
4. Generate test cases from the ECs. You can often use Myers heuristics for combination to generate a minimal set of test cases. Test cases are best documented using a *test case table*.

Applying the process and using the heuristics require some practice. In the remainder of this section I will present some examples of applying both process and heuristics.

34.2.4 Example: Weekday

Consider the following (somewhat contrived) method:

```
public interface weekday {
    /** calculate the weekday of the 1st day of the given month.
     * @param year the year as integer. 2000 means year 2000 etc. Only
     * years in the range 1900–3000 are valid. The output is undefined
     * for years outside this range.
     * @param month the month as integer. 1 means January, 12 means
     * December. Values outside the range 1–12 are illegal.
     * @return the weekday of the 1st day of the month. 0 means Sunday
     * 1 means Monday etc. up til 6 meaning Saturday.
     */
    public int weekday(int year, int month)
        throws IllegalArgumentException;
}
```

Step 1: The conditions are easy to spot. As they are all about ranges, I can apply the range heuristics to get the following equivalence class table.

Condition	Invalid ECs	Valid ECs
year	$< 1900 [y1]; > 3000 [y2]$	$1900 - 3000 [y3]$
month	$< 1 [m1]; > 12 [m2]$	$1 - 12 [m3]$

Using my compact semi-mathematical set notation, you can read the first line like: The requirement for “year” has conditions that lead to an invalid EC, labelled $y1$, having all elements $year < 1900$; an invalid EC, labelled $y2$, having elements $year > 3000$; and a valid EC, labelled $y3$, having elements in the valid range $year \in \{1900..3000\}$.

I use labels like $[y2]$ to make the y remind myself that it is a partitioning of the year condition.

Step 2: What about leap years? I conclude that EC $[y3]$ needs to be repartitioned as e.g. leap year 1976 may not be a good representative for non leap years. So I repartition according to the leap year definition (“Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400.”)

Condition	Invalid ECs	Valid ECs
year (y)		$\{y y \in [1900; 3000] \wedge y \% 400 = 0\} [y3a]$ $\{y y \in [1900; 3000] \wedge y \% 100 = 0 \wedge y \notin [y3a]\} [y3b]$ $\{y y \in [1900; 3000] \wedge y \% 4 = 0 \wedge y \notin [y3a] \cup [y3b]\} [y3c]$ $\{y y \in [1900; 3000] \wedge y \% 4 \neq 0\} [y3d]$

(the table above omits repeating EC $[y1]$, $[y2]$, $[m1]$, $[m2]$, and $[m3]$)

One may also argue that weekday calculation is influenced by leap years with respect to months that are before and after the leap day in February. Thus we may question the representation property of partition $[m3]$. A further division seems worth the effort.

Condition	Invalid ECs	Valid ECs
month		$1 - 2 [m3a]; 3 - 12 [m3b]$

Step 3: The coverage is OK, all possible values of the vector (year, month) belongs to one or another EC.

Step 4: For the test case generation, I can apply Myers heuristics for finding test cases. First cover as many valid partitions as possible; next define one for each invalid one. Thus we end up with the following test cases (I've been a bit lazy about the 'expected' values as this would require consulting the calendars; and I've shortened year to y and month to m).

ECs covered	Test case	Expected output
$[y3a], [m3a]$	$y = 2000; m = 2$	-
$[y3b], [m3b]$	$y = 1900; m = 5$	-
$[y3c], [m3b]$	$y = 2004; m = 10$	5
$[y3d], [m3a]$	$y = 1985; m = 1$	-
$[y1], [m3b]$	$y = 1844; m = 4$	[exception]
$[y2], [m3b]$	$y = 4231; m = 8$	[exception]
$[m1], [y3d]$	$y = 2003; m = 0$	[exception]
$[m2], [y3c]$	$y = 2004; m = 13$	[exception]

Note how one test case covers all valid partitions while we define one test case for each illegal partition. Also note the very important property that only one parameter is illegal at a time to ensure that no masking occurs. Even though we defined quite a few more partitions than in the original proposal (without leap year) it has led to only three extra test cases.

34.2.5 Example: Sum

It is important that you focus on the *requirements* and the conditions associated with them instead of just looking at parameters to a method. Consider the following requirement:

Copyrighted Material. Do Not Distribute.

```
public class PrettyStupid {
    private int T;

    /** return true iff x+y < T */
    public boolean isMoreThanSumOf(int x, int y)
}
```

If I just write the EC table based upon the input parameters then I will end in a situation that leads nowhere:

Condition	Invalid ECs	Valid ECs
x		?
y		?

This is because the ECs are not associated with the input parameters in themselves but on the condition in the specification: $x + y < T$. It is therefore this condition (a boolean condition) that defines the ECs:

Condition	Invalid ECs	Valid ECs
x+y	-	$< T[a1]; \geq T[a2]$

The summation also calls for considering the computation heuristics, so I augment the EC table

Condition	Invalid ECs	Valid ECs
x+y	-	$< T[a1]; \geq T[a2]$
neutral x	-	$= 0[b1]; \neq 0[b2]$
neutral y	-	$= 0[c1]; \neq 0[c2]$

The result is the following tests, remember that generating test cases for the neutral elements ($[b1]$ and $[c1]$) add no value and I therefore avoid it.

ECs covered	Test case	Expected output
$[a1], [b2], [c2]$	$x = 12; y = 23; T = 100$	true
$[a2], [b2], [c2]$	$x = -23; y = 15; T = -10$	false

34.2.6 Example: Formatting

Conditions on the output are just as important as conditions on the input. Consider the following formatting method:

```
/** format a string representing of a double. The string is always
    6 characters wide and in the form ###.##, that is the double is
    rounded to 2 digit precision. Numbers smaller than 100 have '0'
    prefix. Example: 123 -> '123.00'; 2,3476 -> '002.35' etc. If the
    number is larger or equal to 999.995 then '***.**' is output to
    signal overflow. All negative values are signaled with '---.--'
*/
public String format(double x);
```

Here, there are a few partitions on the input x , but there are also several conditions on the output side that define interesting input to present to the method.

Condition	Invalid ECs	Valid ECs
overflow / underflow 2 digit rounding	≥ 999.995 [a1]; < 0.0 [a2]	0.0 – 999.994 [a3] (,00x round up) [b1]; (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ($x \neq 0$) [d1] 'x0' suffix ($x \neq 0$) [d2] exact '.00' suffix [d3]

Note that I here apply my rule of thumb for overflow and negative values. Arguably, these inputs are not invalid, but they will most likely be treated as the first thing in the method and the method will “bail out” once it detects it—and then return without further processing:

```
if (x < 0.0) return "----.--"
[...]
```

The next ECs concern that the rounding works correctly and that the '0' are prefixed ('001.03' and not '1.03') and suffixed ('123.30' and not '123.3') correctly on the output. The test cases could thus look like:

ECs covered	Test case	Expected output
[a1]	1234.456	'***. **'
[a2]	-0.1	'_._'
[a3], [b1], [c1], [d1]	212.738	'212.74'
[a3], [b2], [c2], [d2]	32.503	'032.50'
[a3], [b1], [c3], [d3]	7.995	'008.00'
[a3], [b2], [c4], [d1]	0.933	'000.93'

Note that the first condition (6 characters wide and a “.” in the 4th position) are covered though not mentioned in the EC table.

34.2.7 Example: Chess King

Consider a method to validate if a move from position (c1,r1) to position (c2,r2) on a chess board is valid for a chess king.

```
// Precondition: column, row are within the board
// Precondition: player in turn is not considered
public boolean kingMoveIsValid(char c1, int r1, char c2, int r2);
```

The first thing I do is to look at the conditions (the type of condition given in parentheses):

Copyrighted Material. Do Not Distribute.

1. the distance between the “from” and “to” square $((c1,r1),(c2,r2))$ is exactly one: vertically, horizontally, or diagonally (Range).
2. the king is not in check at the “to” square (Boolean).
3. additionally a king can make a castling move, if
 - (a) the king and the rook in question have never been moved (Boolean).
 - (b) there are no pieces on the row section between the rook and the king (Boolean).
 - (c) the king is not in check (Boolean).
 - (d) the king does not move through a square that is attacked by a piece of the opponent (Boolean).
4. there are no friendly pieces at the “to” square (Boolean).

All conditions are formulated such that if they are true then the move is valid. Note also that the preconditions mean I do not have to consider moving off the board nor moving the opponent’s king. The validation method, as seen from a testing viewpoint, has more input parameters than appears in the parameter list: the state of the board is of course an essential “input” to the method.

Based on the single range and multiple boolean conditions, my first sketch of an EC table looks like (again classifying input leading to “bail out” processing as invalid ECs):

Condition	Invalid ECs	Valid ECs
1. distance (d)	$d = 0$ [$a1$]; $d > 1$ [$a2$];	$d = 1$ [$a3$]
2. not in check at “to”	false [$b1$]	true [$b2$]
3.(a)	false [$c1$]	true [$c2$]
3.(b)	false [$d1$]	true [$d2$]
3.(c)	false [$e1$]	true [$e2$]
3.(d)	false [$f1$]	true [$f2$]
4. no friendly at ‘to’	false [$g1$]	true [$g2$]

Upon reviewing EC [$a3$] I discover that this partitioning will lead to only one test case for the proper distance $d = 1$, for instance a move one square vertically up like (‘f’,5)–(‘f’,6). However, is this single test case really representative for all $d = 1$ moves? No, I think it is not. Consider a programmer pressed for the delivery deadline and thus in a terrible rush—he may just be subtracting the column and row values without considering taking absolute value like e.g.

```
...
if ( r2 - r1 == 1 ) return true;
```

Note that this implementation passes the (‘f’,5)–(‘f’,6) test case. So I repartition [$a3$] to ensure that moves are tested both vertically and horizontally in both directions:

Condition	Invalid ECs	Valid ECs
row distance		-1 [$a3a$]; 0 [$a3b$]; +1 [$a3c$]
column distance		-1 [$a3d$]; 0 [$a3e$]; +1 [$a3f$]

This ensures I generate valid test cases in which the king moves in all directions and both vertically and horizontally. Note that the combination $[a3b]$ $[a3e]$ is of course not possible as the distance is zero.

The start of a test case table for the valid ECs may look like

ECs covered	Test case	Expected output
$[a3a], [a3e], [g2]$	('f',5) to ('f',4)	valid
$[a3b], [a3f], [g2]$	('f',5) to ('g',5)	valid
$[a3c], [a3d], [g2]$	('f',5) to ('e',6)	valid
...		

I now have three test cases for legal moves instead for only one, and these three test cases are guaranteed to test absolute value in the distance calculations in both vertical and horizontal directions. Note that the test cases are formulated terse but there are several underlying assumptions: there is no blocking friendly piece on the 'to' square, and there is indeed a king located on ('f',5).

A further repartitioning concern is “does capture work correctly?” Perhaps a programmer just rejects the move as invalid if any piece is located on the 'to' square, not just a friendly piece? I can repartition the *no friendly at “to” square* condition into a set condition: *contents of “to” square* and update the EC table:

Condition	Invalid ECs	Valid ECs
4. 'to' square contents	friendly $[g1]$	empty $[g2a]$; opponent $[g2b]$

Exercise 34.2: Review the table for more potential repartitioning. Complete the EC and test case tables.

34.3 Boundary Analysis

Experience shows that test cases focusing on *boundary conditions* have a high payoff. Boundaries are usually expressed by conditional statements in the code, and these conditionals are often wrong (the notorious “off by one” error) or plainly missing. Well known examples are for instance iteration over a C array that runs just over the maximal size or calling methods on an object reference that may become null.

Definition: Boundary value

A boundary value is an element that lies right on or next to the edge of an equivalence class.

As an example, consider the chess board position example. As the conditions on row and column are of the range type, the boundaries are explicit: 'a', 'h', 1 and 8, and testing on and just next to these boundaries are very important as conditions can easily be coded incorrectly:

```
if ( row <= 1 ) return false; // should have been row < 1
```

The set and boolean condition guidelines generate ECs that have the boundaries embodied in the ECs themselves: for instance, *true* and *false* are the only valid elements for the boolean condition.

Another example is the case of the formatting method in which there is an abrupt change of behavior at the boundary value of negative values. Thus 0.0 is an interesting value because an error in the condition may return '—.' for this value. The same argument goes for the overflow boundary making 999.94 and 999.95 obviously interesting.

Boundary value analysis is an important tool to detect some of the most difficult defects; but you should always be aware what the underlying implementation actually does. Some algorithms define a “continuum of computation” where an *off by one* error at a boundary will not show up at all. As an example, consider an algorithm to add interest to a bank account: if the balance is above \$0 then a 2% interest rate is added, otherwise an 8% interest rate is deducted. An obvious boundary is of course 0. But—the problem is that we cannot detect any difference in the output no matter what the interest rate calculation is for an account with balance 0. Thus boundary values are less interesting here.

34.4 Discussion

Equivalence partitioning and boundary value analysis are important techniques for finding high quality test cases. Here “high quality” means test cases that have a high probability of finding defects. Still, remember that they are both means to an end, not the end itself. This perhaps obvious fact is however easy to miss once you get absorbed in partitioning, describing ECs using set notation, and combining ECs to produce test cases. Here are some of the pitfalls I often see people fall into.

Key Point: Observe unit preconditions

Do not generate test cases for conditions that a unit specifically cannot or should not handle.

I often see a tendency to generate lots of test cases for invalid ECs and rather few test cases for valid ECs. However, once people start to convert the invalid test cases into automated tests in for instance JUnit they experience that they either cannot express the test case in code or the unit under test is not designed to cope with the input values after all. If a method states that it is a precondition that some input parameter is within a certain range then it does not make sense to generate a test case with a value outside this range. For instance an algorithm to calculate some value based upon a wind direction in degrees may state the precondition that it is within the 0–359 degree range. This is plausible as the hardware simply cannot produce any value outside this range. It is thus a waste of effort to define test cases above 359 and below 0 degrees. (Including them in the EC table is OK as documentation). Another example is graphical user interfaces that often do partial range checks before calling domain units. For instance, a method `move(from, to)` of any board game domain implementation can have as precondition that there is indeed a piece located on the “from” location, because a user cannot grab a non-existing piece using the mouse on a graphical playing board.

Key Point: Systematic testing assumes competent programmers

Equivalence partitioning and other testing techniques rely on honest and competent programmers that are using standard techniques.

Consider a highly incompetent programmer that implements `Math.abs` using an incredible long switch statement:

```
public int abs(int x) {  
    switch(x) {  
        case 1: return 1;  
        case 2: return 2;  
        case 3: return 3;  
        case 7123: return 8222;  
        case -1: return -1;  
        ...  
    }  
}
```

A black box testing technique only considers the specification and this incredibly stupid implementation is hidden from the tester. In this case, of course, the element 7123 from the $EC = \{x | x > 0\}$ is not representative of the other elements, but a tester has no chance of knowing that. Also, black box techniques cannot discover “easter eggs” or real malicious code fragments that are triggered by special input values, combinations of them or special sequences. Other techniques, notably **systematic reviews**, are required to find such issues.

Key Point: Do not use Myers combination heuristics blindly

Myers heuristics for generating test cases from valid and invalid ECs can lead to omitting important test cases.

Many algorithms simply return the result of a complex calculation but with very few special cases and thus the specification contains few or no “conditions”. Focusing on conditions in the EC finding process is good at treating the special cases but the combination rule that merge as many valid ECs as possible sometimes leads to a situation where just zero or a single test case are defined for the complex computation! Of course, test cases must also be defined for the ordinary computations.

34.5 Summary of Key Concepts

Systematic testing is a systematic process for finding defects in some unit under test. Testing techniques are generally considered either *black-box* or *white-box*. In black-box techniques you only consider the specification and common knowledge of programming while in white-box testing you in addition inspect the source code. Many systematic testing techniques require a substantial effort and is thus best used on complex and/or essential software units.

The *equivalence class partitioning* technique is based on partitioning the input space into *equivalence classes* which are subsets of the input space. Elements in an equivalence class (EC) must all have the *representation* property, that is, if one element in the EC exposes a defect during testing, then all other elements in the EC must expose

the same defect. The set of all ECs must *cover* the full input set for the set of ECs to be considered sound. Finding ECs is an iterative and heuristic process: a good starting point is *conditions* in the specification of the unit. Guidelines for *ranges*, *sets*, and *boolean conditions* allow a first partitioning to be found. Regarding *computations* you must partition on neutral elements. These partitions should be closely reviewed to ensure the representation property, if in doubt, then ECs should be further repartitioned. Next, test cases are generated by combining ECs. One approach is to make every combination of ECs. This, however, often leads to a very high number of test cases. Another approach is that of Myers in which as many valid ECs are covered as possible until the valid ECs are exhausted; and next invalid ECs are covered in which only one element is taken from an invalid EC at the time. The latter rule is important to avoid *masking*: the ability for the error checking code for one parameter to mask a defect in the error checking code for another parameter.

Finally, *boundary analysis* complements EC testing as it identifies the values right on or next to edges of ECs as particularly important to test as experience has shown that programmers often have “off by one” errors in their conditions. Boundary analysis is important for range conditions but is often not applicable for other types of conditions.

The treatment in this chapter is inspired by heuristics first outlined by Myers (1979) and later restated by Burnstein (2003). Binder (2000) is a comprehensive book on testing object-oriented software.

34.6 Selected Solutions

Discussion of Exercise 34.1:

A) Removing a defect in *dead* code (i.e. code that can never be executed) does not change the software’s reliability.

B) As an example, if there is a single defect in an application’s “save” feature it may make the application useless; contrast this to a defect in some obscure, seldom used, function of the software.

34.7 Review Questions

Define the concepts *systematic testing*, *black-box* and *white-box testing*.

Define what an *equivalence class* is and name the properties that must be fulfilled for the set of ECs to be sound. Why is partitioning the input space into ECs interesting?

Describe the process of finding test cases using the EC technique: what steps are involved? What guidelines are used to find ECs? What guidelines are used to construct test cases based on the ECs. Why is ECs often repartitioned further?

Define *boundary value analysis* and explain what it is.

34.8 Further Exercises

Exercise 34.3:

Generate test cases using the equivalence class partitioning technique for the method to validate moves in Breakthrough, defined in exercise ??.

1. Outline the EC table. Describe the analysis in terms of the guidelines applied and argue for *coverage* and *representation*.
2. Outline the test case table, and argue how the test cases have been generated.

Exercise 34.4:

Add test cases to your test case table for Breakthrough move validation from the previous exercise based on a boundary value analysis.

Exercise 34.5:

The MET REPORT encodes local airport weather information in an internationally accepted format. The MET REPORT is broadcasted locally in the airport to flight leaders and forwarded to incoming aircrafts. To a pilot the characteristics of the wind on the landing strip is important and a MET REPORT contains a 5 character field denoted `dddff` that provides wind data. For instance, if the field contains “09012” then a 12 knot wind blows in direction east and “00005” is a 5 knot wind blowing north. The first three digits (`ddd`) code the *wind direction* in degrees [0;359] where 0 is north, 90 is east, etc. The last two digits (`ff`) code the *wind speed* in knots. Both `ddd` and `ff` are calculated based upon the *two minute mean wind*, that is, the wind direction and speed is sampled every 10 seconds and a floating mean is calculated over the samples from the last two minutes. A 2 minute mean measurement contains also the upper and lower extremes for the wind direction. For example the mean wind direction may be 90, the low extreme 60 and the high extreme 200; this would show that the prevailing wind has been east but there seems to have been a sudden, short, change towards south.

You can consider that the method to calculate the string representing `dddff` is defined by a method:

```
public String dddff( TwoMinuteMeanWind w )
```

that takes an object, `w`, representing the two minute mean wind coded as a simple data object:

```
public class TwoMinuteMeanWind {
    /** true iff sensor readings are valid */
    public boolean valid;
    /** two minute mean wind direction; the value
        is between 0 and 359 degrees. */
    public int direction;
    /** two minute mean wind speed in knots; the
        value is between 0.0 and 99.9 */
}
```

```
public double speed;
/** low extreme wind direction in the
    two minute time span; range [0;359] degrees.
    PostCondition: low < high always. */
public int low;
/** high extreme wind direction in the
    two minute time span; range [0;359] degrees.
    PostCondition: low < high always. */
public int high;
}
```

Note that 'low' is always the numerical lowest value, that is it is always true that 'low' < 'high'. This data object is guaranteed to be correct by the wind measuring hardware and obey all post conditions and specifications.

The output string of `dddff` is defined as follows:

1. The output `dddff` string is always a full 5 character string. If a direction or speed is not three digits or two digits respectively, then '0's are prefixed. That is, direction 7 degrees and speed 2 knots is coded "00702".
2. However, if the sensor is broken ('w.valid'==false), no samples are provided and `dddff` is coded as "**** " indicating 'not known'. (Note the space character at the end.)
3. If the wind speed is below 0.5 knots, then the `dddff` output string is "CALM " no matter what the wind direction is. (Again, note the space at the end.)
4. If the difference between the lower and upper extremes of wind direction is more than 180 degrees then the `ddd` is coded as "VRB" meaning *variable wind*. Note that you cannot simply subtract 'high' – 'low' to find if it is a VRB condition; you have to take the wind direction into account. (Example: low = 80 and high = 280 is not a VRB condition if `ddd`=10; but it is a VRB condition if `ddd`=180.)
5. CALM takes precedence over VRB (the wind sensor simply does not move in CALM conditions thus the lower and upper extremes are simply non-sense.)

Generate test cases using the equivalence class partitioning technique for the method `dddff`.

1. Outline the EC table. Describe the analysis in terms of the guidelines applied and argue for *coverage* and *representation*.
2. Outline the test case table, and argue how the test cases have been generated.

Exercise 34.6:

Add test cases to your test case table for `dddff` from the previous exercise based on a boundary value analysis.