Software Testing

Lab Assignments

March 10, 2025

Niels Van der Planken
s0191930@ad.ua.ac.be

Thomas Gueutal
s0195095@ad.ua.ac.be

Contents

1 Introduction

This report is a deliverable for one of the testing assignments for the course Software Testing (Master Computer Science 2001WETSWT) at the University of Antwerp in the academic year of 2024-2025.

As per the course description, "*The objective of the lab work of the Software Testing course is to help you learn how you can apply the various testing techniques and test design patterns as discussed during the lectures in practice. You will apply these techniques to a simple Pacman system written in Java. The amount of coding that needs to be done is relatively small: The focus is on testing.*".

This report was written in **Typst**.

2 Division of Tasks

The deliverables are as follows.

1. Code solutions to the tasks, located at **this** Github repo. Note that we add the source code at the end of an assignment as a separate release to the github repo.
2. This report, discussing the solutions and code.

We briefly discuss the division of tasks. We should strive for a half-half split of work between the two team members.

Member	Tasks
Niels	Exercises 1 - 4, 10 - 11 (including report)
Thomas	Exercises 1 - 4, 10 - 11 (including report)

3 Assignment 02 – Category Partitioning and Boundary Values

We repeat a note made at the start of the assignment file.

> **Important Note:** Create an archive for JPacman system after performing all the exercises that require modifications to the files. Submit it along with your report. Refer to the "Assignments General" document for introductory information.

"*In this assignment, you will be building and testing an implementation* `Board.withinBorders(int x, int y)` *method, which simply checks that* x *and* y *integer values fall within the borders (width* $\times$ *height) of the board, i.e.* $0 \leq x < width$*, and* $0 \leq y < height$*. Our approach follows the approach from Forgács (Practical test design : selection of traditional and automated test design techniques), chapter 5: p57-89*" as per the assignment description.

Next, we quote the definitions for ON, OFF, IN and OUT points from the textbook [1].

- A test input on the closed boundary is called an **ON point**; for an open boundary, an ON point is a test input 'closest' to the boundary inside the examined domain.
- A test input inside the examined domain ('somewhere in the middle') is called an **IN point**.
- A test input outside a closed boundary and 'closest' to the ON point is called an **OFF point**; for an open boundary, an OFF point is on the border.
- A test input outside the boundary of the examined domain is called an **OUT point**.

The ON and OFF points have to be 'as close as possible'. This means that if an EP contains only integers, then the distance of the two points is one; for example, if an EP contains book prices, where the minimum price difference is EUR 0.01, then the distance between the points is also EUR 0.01.

3.1 Exercise 1

> **Question:** List at least three possible errors that an implementor of this method could make.

This question is about finding possible sources of faults; which discrepancies between the specification and the implementation of `Board.withinBorders` could possibly be introduced by the programmer.

We look to chapter 5 in *Practical test design* [1] for some inspiration. The section *Fault models in BVA* specifies two types of fault models: *Predicate faults* and *Data (Variable, Operator) faults*. Of course, the use of BVA, Boundary Value Analysis, assumes EP, Equivalence Partitioning, is also used when designing relevant test cases.

1. Predicate fault: A predicate fault refers to incorrectly implementing a (boolean) predicate from how it was phrased in the requirements. This emphasizes that the logical structure of the implemented expression should match that of the one in the specification.
    - $0 \leq x \leq width \land 0 \leq y \leq height$, where we mistakenly use only $\leq$ operators, instead of using $<$ for the upper bounds.
    - $0 \leq x < width \lor 0 < y \leq height$, where the expression is a disjunction (or, $\lor$) instead of a conjunction (and, $\land$).
2. Data fault: A data (variable, operator) fault refers to incorrectly handling the actual variables or program state. This emphasizes program state mismanagement.
    - $0 \leq height < width \land 0 \leq y < height$, where we use the "unrelated" variable *height* instead of the intended variable x. By unrelated we mean that *height* does not serve the role of an input variable; *height* serves the role of a boundary value.
    - $0 \leq x < height \land 0 \leq y < width$, where the use of *width* and *height* was incorrectly swapped. Both *width* and *height* are boundary values, but swapping them results is a mismatch with regards to the specification.
    - $0 < (x = width)$, which incorrectly uses an assignment instead of $\leq$ to implement the partial predicate $0 < x \leq width$. Note that this is a language agnostic

fault, which incidentally can also occur in Java. See Figure 5. The test case `testWithinBorderSucceed` **succeeds**, even though the predicate is not satisfied since $x = 4 \not< 2 = width$. The test case `testWithinBorderFail` **fails** due to a type error; Java's type system somewhat prevents this type of errors in case the predicate is properly implemented as in `0 < x && x <= width`.

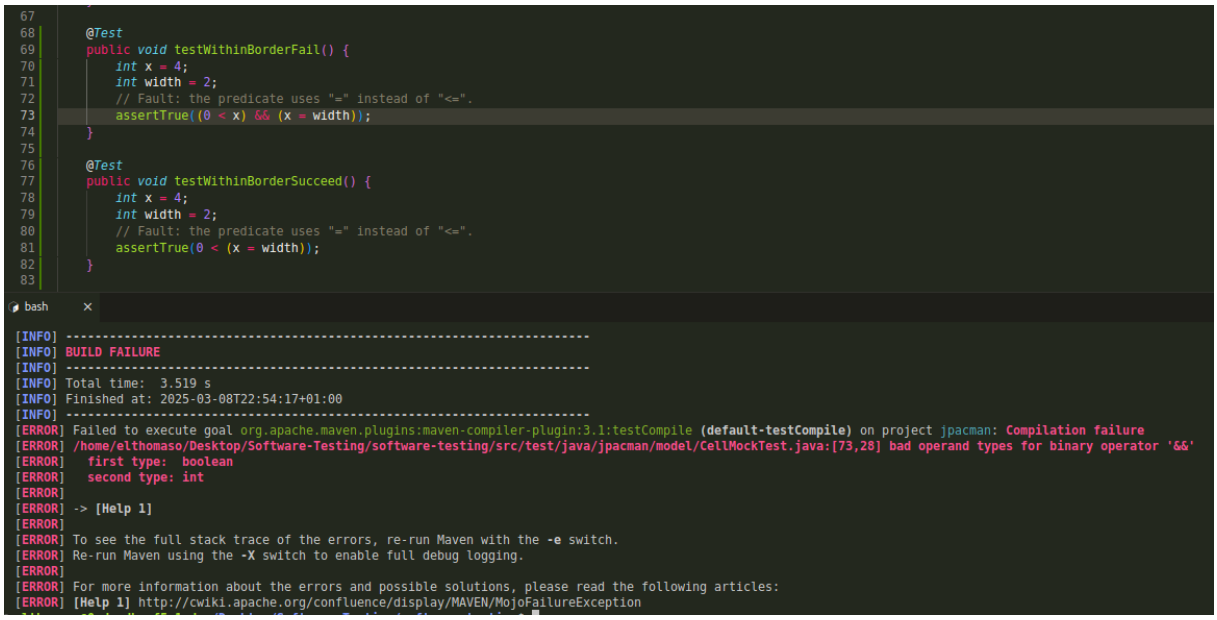

Figure 1: Java's strict type system prevents assignment faults in a proper predicate, but silly mistakes can still happen: `0 < (x = width)`.

3.2 Exercise 2

Question: Identify the equivalence partitions for the *withinBorders* method. You should provide a table similar to Table 5.1 (see Forgács [1], or see Figure 4 in our report). You may assume the *single fault model*.

We briefly repeat the specification; the `Board.withinBorders(int x, int y)` method simply checks that $0 \leq x < width$, and $0 \leq y < height$. The method takes **two independent variables** $-2147483648 \leq x, y \leq 2147483647$ which are fixed size (bounded) integers.

Two test values a, b are part of the same equivalence class iff. the code tests the same behavior (computation) of the test object for both of the inputs a, b. We assume the **single fault model**, which means we only consider partitions that can reveal *at most one fault*. For example, we do *not* consider a partition P with the two independent attributes $x < 0$ and $y < 0$, because if we generate a test case with as data point $(x = -1, y = -1) \in P$, and that test case then fails, then both of the tested, independent variables x, y are potential fault sources. The single fault model dictates that a test case should reveal a singular faulty attribute. The *single fault model* tries to alleviate this limitation: if multiple attributes are faulty at the same time, then correspondingly multiple test cases will fail to reveal the faults individually.

In Table 1 we define the partitions. We visually present them in Figure 2.

Partition #1: The **correct partition**, containing inputs that make the predicate true.
Partition #2: An invalid partition, which checks that the **lower bound of x** is correctly implemented.
Partition #3: An invalid partition, which checks that the **upper bound of x** is correctly implemented.
Partition #4: An invalid partition, which checks that the **lower bound of y** is correctly implemented.
Partition #5: An invalid partition, which checks that the **upper bound of y** is correctly implemented.
Partition #6: The partition containing **all remaining data points**, which violate the single fault model.

Adding this partition is necessary because the partitions $1 - 5$ do not fully partition the entire input domain. This can be clearly seen in Figure 2, where the white regions constitute this final partition.

Equivalence Partitions	Input Attributes	
1 (correct)	$0 \leq x < width$	$0 \leq y < height$
2	$x < 0$	$0 \leq y < height$
3	$width \leq x$	$0 \leq y < height$
4	$0 \leq x < width$	$y < 0$
5	$0 \leq x < width$	$height \leq y$
6	Inputs outside all of the partitions above	

Table 1: Single fault equivalence partitions. Gray shaded cells "*represent the error-revealing partitions assuming the single fault model. If a test fails, then the bug relates to exactly one of the partitions*".

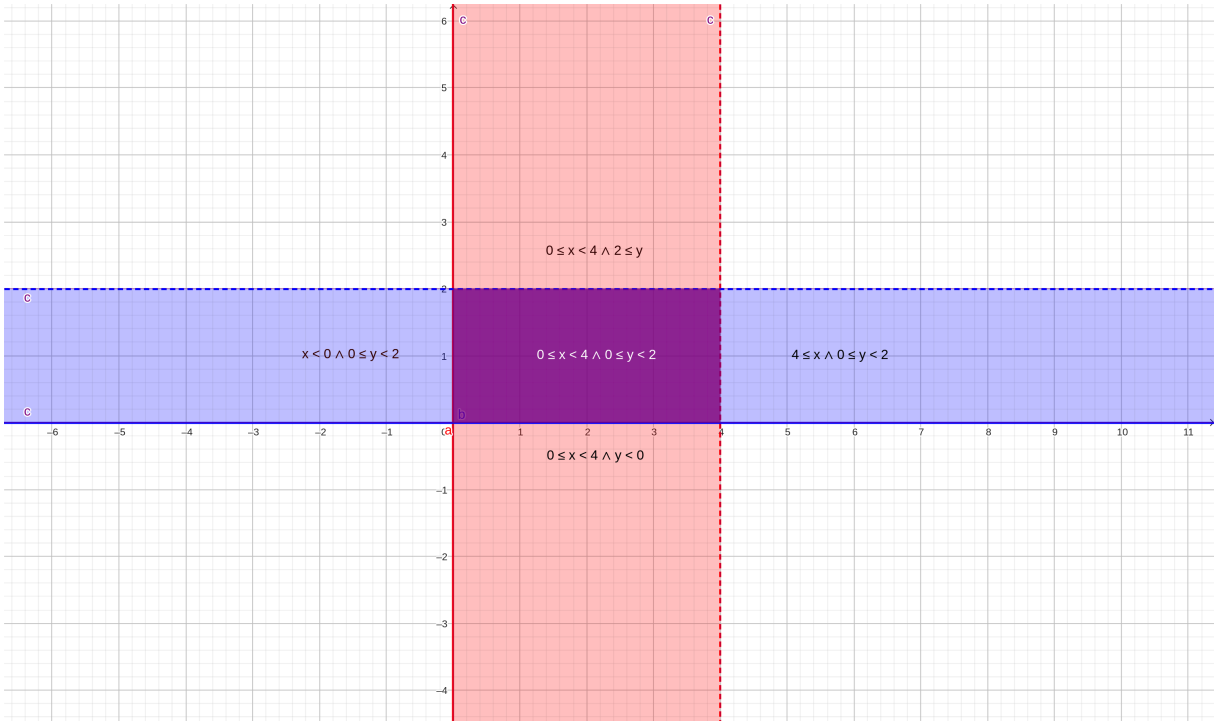

Figure 2: Just for the purpose of plotting, **fix $width = 4$ and $height = 2$**. This plot visually represents the single fault model Equivalence Partitions. Each partition is annotated by its input attributes (its boolean condition). Dotted (full) partition edges are open (closed) boundaries.

3.3 Exercise 3

Question: Similarly, identify the equivalence partitions for the *withinBorders* method, but for the *multiple fault model*.

We briefly repeat the specification; the `Board.withinBorders(int x, int y)` method simply checks that $0 \leq x < width$, and $0 \leq y < height$. The method takes **two independent variables** $-2147483648 \leq x, y \leq 2147483647$ which are fixed size (bounded) integers.

We assume the **multiple fault model**, which means we consider partitions that can reveal *zero or more faults*. For example, we *do* consider a partition P with the two independent attributes $x < 0$ and $y < 0$, because if we generate a test case with as data point $(x = -1, y = -1) \in P$, and that test case then fails, then both of the tested, independent variables x, y are potential fault sources.

In Table 2 we define the partitions. We visually present them in Figure 3.

Partition #1: The **correct partition**, containing inputs that make the predicate true.
Partition #2: An invalid partition, which checks that the **lower bound of x** is correctly implemented.
Partition #3: An invalid partition, which checks that the **upper bound of x** is correctly implemented.
Partition #4: An invalid partition, which checks that the **lower bound of y** is correctly implemented.
Partition #5: An invalid partition, which checks that the **upper bound of y** is correctly implemented.
Partition #6: An invalid partition, which checks that the **lower bound of x and the lower bound of y** are **simultaneously** correctly implemented.
Partition #7: An invalid partition, which checks that the **lower bound of x and the upper bound of y** are **simultaneously** correctly implemented.
Partition #8: An invalid partition, which checks that the **upper bound of x and the lower bound of y** are **simultaneously** correctly implemented.
Partition #9: An invalid partition, which checks that the **upper bound of x and the upper bound of y** are **simultaneously** correctly implemented.

Equivalence Partitions	**Input Attributes**	
1 (correct)	$0 \leq x < width$	$0 \leq y < height$
2	$x < 0$	$0 \leq y < height$
3	$width \leq x$	$0 \leq y < height$
4	$0 \leq x < width$	$y < 0$
5	$0 \leq x < width$	$height \leq y$
6	$x < 0$	$y < 0$
7	$x < 0$	$height \leq y$
8	$width \leq x$	$y < 0$
9	$width \leq x$	$height \leq y$

Table 2: Multiple fault equivalence partitions. Gray shaded cells represent the error-revealing partitions assuming the multiple fault model. If a test fails, then the bug relates to at least one of the partitions.

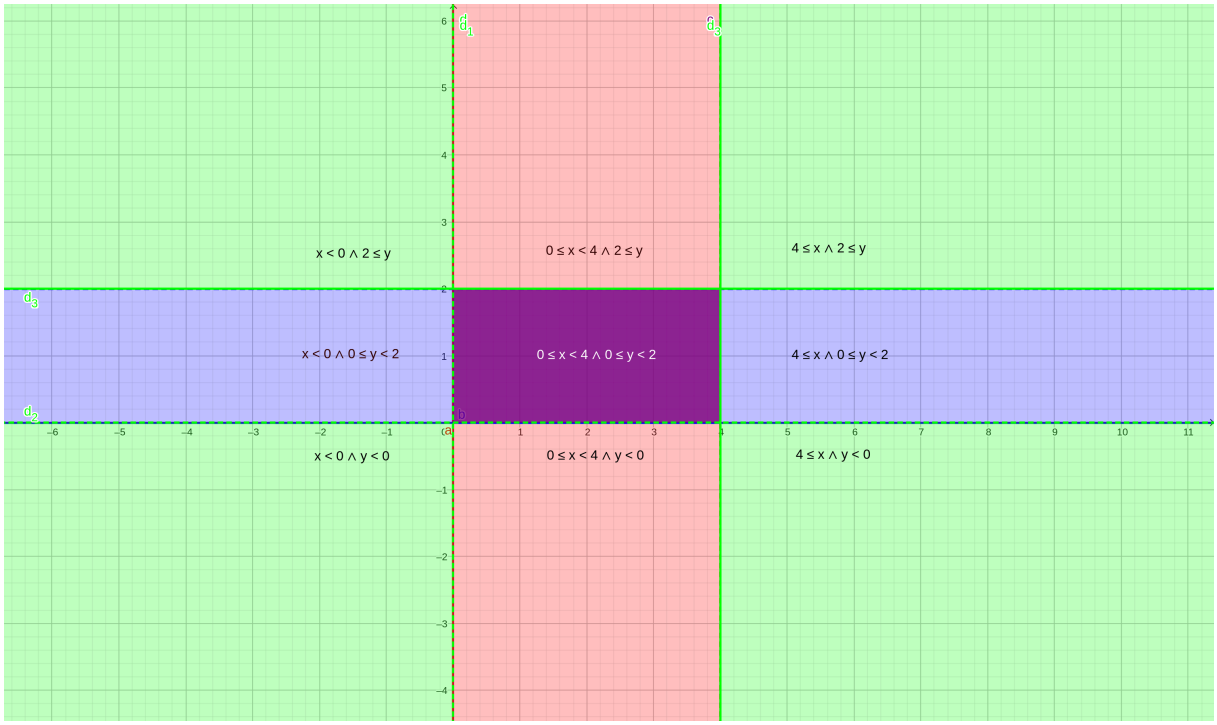

Figure 3: Just for the purpose of plotting, **fix $width = 4$ and $height = 2$**. This plot visually represents the multiple fault model Equivalence Partitions. Each partition is annotated by its input attributes (its boolean condition). Dotted (full) partition edges are open (closed) boundaries.

3.4 Exercise 4

Question: Next, develop a test design against predicate faults (see Forgács, Table 5.2 [1], or see Figure 5 in our report). You should have a table for predicate x and a table for predicate y.

We make the following assumptions.

- $width > 0$, which follows immediately from $0 \leq x < width$.
- $height > 0$, which follows immediately from $0 \leq y < height$.

The goal of test design against predicate faults based on Boundary Value Analysis is to determine which data points around the partition boundaries allow us to detect faults in the implementation of a predicate w.r.t. its specification. In other words, for all possible ways we could incorrectly implement a predicate P, we want there to be at least one test in the test suite that will fail if the predicate is implemented in that incorrect way.

See Table 4, which shows test design against predicate faults for the two predicates $0 \leq x$ and $x < width$. It specifies which BVA data points will uncover a fault (gray shaded cells) for each possible (incorrect) implementation of either of the two predicates. For the predicate $0 \leq x$ the data points -1 (OFF), 0 (ON) and 10 (IN) cover all possible faulty implementations, since every row has at least on gray shaded cell. For the predicate $x < width$ the data points *width* (OFF), $width - 1$ (ON) and $width + 10$ (OUT) cover all possible faulty implementations.

See Table 3, which similarly shows test design against predicate faults for the two predicates $0 \leq y$ and $y < height$. This table is virtually identical to that of x, since the predicates are similar as well. For the predicate $0 \leq y$ the data points -1 (OFF), 0 (ON) and 10 (IN) cover all possible faulty implementations. For the predicate $y < height$ the data points *height* (OFF), $height - 1$ (ON) and $height + 10$ (OUT) cover all possible faulty implementations.

Note that there can be overlap between the test data points that cover fault detection of different predicates. For example, the IN point of predicate $0 \leq x$ could be chosen to be the same as the OFF point of predicate $x < width$, given the right choice of *width*. This way you can minimize the amount of test cases that have to be implemented, while still guarding against all possible faulty predicate implementations.

Program version no.	**Correct/Wrong predicate**	**Test data 1**	**Test data 2**	**Test data 3**
	x	**Specific values of the variable x**		
		-1 (OFF)	0 (ON)	10 (IN)
			Output	
1 (correct)	$0 \leq x$	F	T	T
2	$0 < x$	F	F	T
3	$0 \geq x$	T	T	F
4	$0 > x$	T	F	F
5	$0 = x$	F	T	F
6	$0 \neq x$	T	F	T
7	$1 \leq x$	F	F	T
8	$-1 \leq x$	T	T	T
	x	**Specific values of the variable x**		
		width (OFF)	$width - 1$ (ON)	$width + 10$ (OUT)
			Output	
1 (correct)	$x < width$	F	T	F
2	$x \leq width$	T	T	F
3	$x > width$	F	F	T
4	$x \geq width$	T	F	T
5	$x = width$	T	F	F
6	$x \neq width$	F	T	T
7	$x < width + 1$	T	T	F
8	$x < width - 1$	F	F	F

Table 3: Test design against predicate faults. BVA for predicates $0 \leq x$ and $x < width$. Gray shaded cells represent a fault being uncovered in the implemented predicate when the given input is used as a test value.

Program version no.	Correct/Wrong predicate	Test data 1	Test data 2	Test data 3
	y	Specific values of the variable y		
		-1 (OFF)	0 (ON)	10 (IN)
		Output		
1 (correct)	$0 \leq y$	F	T	T
2	$0 < y$	F	F	T
3	$0 \geq y$	T	T	F
4	$0 > y$	T	F	F
5	$0 = y$	F	T	F
6	$0 \neq y$	T	F	T
7	$1 \leq y$	F	F	T
8	$-1 \leq y$	T	T	T
	y	Specific values of the variable y		
		height (OFF)	$height - 1$ (ON)	$height + 10$ (OUT)
		Output		
1 (correct)	$y < height$	F	T	F
2	$y \leq height$	T	T	F
3	$y > height$	F	F	T
4	$y \geq height$	T	F	T
5	$y = height$	T	F	F
6	$y \neq height$	F	T	T
7	$y < height + 1$	T	T	F
8	$y < height - 1$	F	F	F

Table 4: Test design against predicate faults. BVA for predicates $0 \leq y$ and $y < height$. Gray shaded cells represent a fault being uncovered in the implemented predicate when the given input is used as a test value.

3.5 Exercise 5

Question: Implement your test case specs into *BoardTest* class and run the test suite.

To be done.

3.6 Exercise 6

Question: Implement *withinBorders* method. Then re-run the test suite. Inspect code coverage results, and explain your findings.

To be done.

3.7 Exercise 8

Question: Would your test suite reveal all the faults you proposed in Exercise 1? If not, explain why the equivalence partitioning approach missed it, and add appropriate test cases separately to your JUnit implementation.

To be done.

3.8 Exercise 10

Question: Repeat the exercises for the method *Game.addGuestFromCode(char code, int x, int y)*. How many test cases do you end up with?

First, let us analyze the `Game.addGuestFromCode` method. See Codeblock 3 for the source code.

The assertions of the method detail its pre-conditions and post-conditions; we can derive the method's specification from its assertions and implementation. We determine the relevant aspects of the specification, which are affected only by the input parameters $code, x, y$. We do not look at predicates that depend solely on any class members, since we can not influence their initial state through the input parameters of a given function call.

- The pre-conditions $0 \leq x < width$ and $0 \leq y < height$ both follow from the pre-condition assertion `assert getBoard().withinBorders(x, y);` and consequently from the specification of `Board.withingBorders(int x, int y)` .
- There is a **finite set of valid values** for the ***code* variable**, see Table 5. Let us **call this set** $\mathbb{V} = \{'0', 'F', 'M', 'P', 'W'\} = \{48, 70, 77, 80, 87\}$. Furthermore, the switch statement enumerates each of these valid types, and throws an assertion error if a *code* value different from those in Table 5 (i.e. $code \notin \mathbb{V}$) is found in the default switch case. We state that $\boldsymbol{code \in \mathbb{V}}$ **is a requirement**.

Name	Character	ASCII Code
EMPTY_TYPE	'0'	48
FOOD_TYPE	'F'	70
MONSTER_TYPE	'M'	77
PLAYER_TYPE	'P'	80
WALL_TYPE	'W'	87

Table 5: The valid `Guest` types.

3.8.1 Equivalence Partitioning

We first determine the **single fault model** based partitions. We note that such partition definitions are purely dependent on the input parameter domains. We note three independent variables $code, x, y$ with the following requirements.

- $0 \leq x < width$
- $0 \leq y < height$
- $code \in \mathbb{V}$

See Table 6 for the single fault partitions. The only difference from the single fault partitions in Table 1 from before, is the addition of the *code* parameter. We added a single new partition to account for the invalid *code* subdomain. Visually, the space is partitioned into five instances of Figure 2, each at a different z coordinate corresponding to one of the five valid Guest type ASCII codes in Table 5, and a final partition that contains all remaining, un-partitioned inputs.

Equivalence Partitions	Input Attributes		
1 (correct)	$code \in \mathbb{V}$	$0 \leq x < width$	$0 \leq y < height$
2	$code \notin \mathbb{V}$	$0 \leq x < width$	$0 \leq y < height$
3	$code \in \mathbb{V}$	$x < 0$	$0 \leq y < height$
4	$code \in \mathbb{V}$	$width \leq x$	$0 \leq y < height$
5	$code \in \mathbb{V}$	$0 \leq x < width$	$y < 0$
6	$code \in \mathbb{V}$	$0 \leq x < width$	$height \leq y$
7	Inputs outside all of the partitions above		

Table 6: Single fault equivalence partitions. Gray shaded cells "*represent the error-revealing partitions assuming the single fault model. If a test fails, then the bug relates to exactly one of the partitions*".

Next, see **Table 7** for the multiple fault partitions. Again, we adapted **Table 2** from before, by taking the *code* parameter into account. We add many additional partitions to account for combinations with the valid and invalid *code* subdomains.

Equivalence Partitions	Input Attributes		
1 (correct)	$code \in \mathbb{V}$	$0 \leq x < width$	$0 \leq y < height$
2	$code \notin \mathbb{V}$	$0 \leq x < width$	$0 \leq y < height$
3	$code \in \mathbb{V}$	$x < 0$	$0 \leq y < height$
4	$code \in \mathbb{V}$	$width \leq x$	$0 \leq y < height$
5	$code \in \mathbb{V}$	$0 \leq x < width$	$y < 0$
6	$code \in \mathbb{V}$	$0 \leq x < width$	$height \leq y$
7	$code \in \mathbb{V}$	$x < 0$	$y < 0$
8	$code \in \mathbb{V}$	$x < 0$	$height \leq y$
9	$code \in \mathbb{V}$	$width \leq x$	$y < 0$
10	$code \in \mathbb{V}$	$width \leq x$	$height \leq y$
11	$code \notin \mathbb{V}$	$0 \leq x < width$	$y < 0$
12	$code \notin \mathbb{V}$	$0 \leq x < width$	$height \leq y$
13	$code \notin \mathbb{V}$	$0 \leq x < width$	$y < 0$
14	$code \notin \mathbb{V}$	$0 \leq x < width$	$height \leq y$
15	$code \notin \mathbb{V}$	$x < 0$	$0 \leq y < height$
16	$code \notin \mathbb{V}$	$x < 0$	$0 \leq y < height$
17	$code \notin \mathbb{V}$	$width \leq x$	$0 \leq y < height$
18	$code \notin \mathbb{V}$	$width \leq x$	$0 \leq y < height$
19	$code \notin \mathbb{V}$	$x < 0$	$y < 0$
20	$code \notin \mathbb{V}$	$x < 0$	$height \leq y$
21	$code \notin \mathbb{V}$	$width \leq x$	$y < 0$
22	$code \notin \mathbb{V}$	$width \leq x$	$height \leq y$

Table 7: Multiple fault equivalence partitions. Gray shaded cells represent the error-revealing partitions assuming the multiple fault model. If a test fails, then the bug relates to at least one of the partitions.

3.8.2 BVA Test Design Against Predicate Faults

In the Boundary Value Analysis we **only focus on predicate faults**, and **omit any data** (variable, operator) **fault analysis**. This is because the preceding exercises did not require any variable fault analysis.

First of all, we refer to the predicate fault analysis presented in Section 3.4 for the **four predicates** $0 \leq x$, $x < width$, $0 \leq y$ and $y < height$. The boundary value test data points we suggested there are also valid for testing the `Game.addGuestFromCode(char code, int x, int y)` method, because the x, y parameters of this method must adhere to the same requirements as stated for `Board.withinBorders(int x, int y)`. If we suppose all three data points for each of the four predicates are turned into test cases, then **they contribute twelve test cases**.

Second, you could consider each case in the switch statement to be an individual equality predicate. This follows from the fact that a switch statement where every case has a `break` is semantically equivalent to an if-else chain, see Table 8. **The benefit of the switch statement** is that with regards to predicate fault analysis, it is **impossible to incorrectly implement the `==` operator**. So the predicates implied by the switch do not have to be analyzed here.

```
switch (code) {
case WALL_TYPE:
    break;
case PLAYER_TYPE:
    break;
case FOOD_TYPE:
    break;
case MONSTER_TYPE:
    break;
case EMPTY_TYPE:
    break;
default:
    break;
}
```

```
if (code == WALL_TYPE) {
  ...
} else if (code == PLAYER_TYPE) {
  ...
} else if (code == FOOD_TYPE) {
  ...
} else if (code == MONSTER_TYPE) {
  ...
} else if (code == EMPTY_TYPE) {
  ...
// Default case.
} else {
  ...
}
```

Next we examine the predicates found after the switch statement, see Codeblock 1. Note that we may rewrite the ternary operator used in the first assert to be a boolean predicate, see Codeblock 2.

As a side note, if we may rely on the correct implementation of the switch case, then the predicate $theGuest == null$ can be rewritten as $code == Guest.EMPTY\_TYPE$ or equivalently $code = 48$, because the switch case for $code = Guest.EMPTY\_TYPE$ sets `theGuest = null;`. Either way, we conclude that the only remaining predicate that we may influence by manipulating the input parameters is $code == Guest.EMPTY\_TYPE$ which we rephrase as $code = 48$ to be able to use the ASCII code value, and for convenience.

```
assert code == Guest.EMPTY_TYPE ? theGuest == null : theGuest != null;

if (theGuest != null) {
    theGuest.occupy(getBoard().getCell(x, y));
}

assert theGuest == null ||
       getBoard().getCell(x, y).equals(theGuest.getLocation());
```

Codeblock 1: The source code that follows the switch statement, see Codeblock 3.

```
// The original assertion condition.
    code == Guest.EMPTY_TYPE ?
        theGuest == null :
        theGuest != null
// Rewrite the condition as a predicate.
=   (code == Guest.EMPTY_TYPE AND theGuest == null) OR
    (code != Guest.EMPTY_TYPE AND theGuest != null)
=   (code == Guest.EMPTY_TYPE) == (theGuest == null)
```

See Table 9 for the test design against predicate faults. Only three BVA test cases suffice to cover all possible incorrect implementations of the predicate $code = 48$.

Program version no.	**Correct/Wrong predicate**	**Test data 1**	**Test data 2**	**Test data 3**
	$code$	**Specific values of the variable y**		
		48 (ON)	47 (OFF)	49 (OFF)
		Output		
1 (correct)	$code = 48$	T	F	F
2	$code > 48$	F	F	T
3	$code \geq 48$	T	F	T
4	$code < 48$	F	T	F
5	$code \leq 48$	T	T	F
6	$code \neq 48$	F	T	T
7	$code = 49$	F	F	T
8	$code = 47$	F	T	F

Table 9: Test design against predicate faults. BVA for predicate $code = 48$. Gray shaded cells represent a fault being uncovered in the implemented predicate when the given input is used as a test value.

3.9 Exercise 11

Question: Imagine a more complex *Board* and an additional parameter that describes the shape of the *Guest* (like occupying multiple cells). How does an equivalence partitioning approach scale?

Table 7 gives us the impression that the number of partitions under the **multiple fault model** assumption will increase fairly quickly by:

1. adding additional input parameters, because there are more fault combinations to consider.
2. introducing additional complexity into the variable domain definitions, because there are more unique partitions to consider due to the increase in the number of distinct domain boundaries.

```java
    /**
     * Add a new guest to the board.
     * @param code Representation of the sort of guest
     * @param x x-position
     * @param y y-position
     */
    private void addGuestFromCode(char code, int x, int y) {
        assert getBoard() != null : "Board should exist";
        assert getBoard().withinBorders(x, y);
        Guest theGuest = null;
        switch (code) {
        case Guest.WALL_TYPE:
            theGuest = new Wall();
            break;
        case Guest.PLAYER_TYPE:
            theGuest = createPlayer();
            break;
        case Guest.FOOD_TYPE:
            theGuest = createFood();
            break;
        case Guest.MONSTER_TYPE:
            theGuest = createMonster();
            break;
        case Guest.EMPTY_TYPE:
            theGuest = null;
            break;
        default:
            assert false : "unknown cell type``" + code + "'' in worldmap";
        break;
        }
        assert code == Guest.EMPTY_TYPE ? theGuest == null : theGuest != null;
        if (theGuest != null) {
            theGuest.occupy(getBoard().getCell(x, y));
        }
        assert theGuest == null
        || getBoard().getCell(x, y).equals(theGuest.getLocation());
    }
```

Codeblock 3: The source code of the `Game.addGuestFromCode(char, int, int)` method.

Table 5.1 Equivalence partitioning for the authorisation example

Equivalence partitions	Password attributes				
1	Number of characters >= 8 and <= 14	At least one lower case character	At least one upper case character	At least one numeric character	At least one character: ':', ';', '<', '=', '>', '?', '@'
2	**Number of characters < 8**	At least one lower case character	At least one upper case character	At least one numeric character	At least one character: ':', ';', '<', '=', '>', '?', '@'
3	**Number of characters > 14**	At least one lower case character	At least one upper case character	At least one numeric character	At least one character: ':', ';', '<', '=', '>', '?', '@'
4	Number of characters ≥ 8 and ≤ 14	At least one lower case character	At least one upper case character	At least one numeric character	**No special character: ':', ';', '<', '=', '>', '?', '@'**
5	Number of characters ≥ 8 and ≤ 14	At least one lower case character	At least one upper case character	**No numeric character**	At least one character: ':', ';', '<', '=', '>', '?', '@'
6	Number of characters ≥ 8 and ≤ 14	At least one lower case character	**No upper case character**	At least one numeric character	At least one character: ':', ';', '<', '=', '>', '?', '@'
7	Number of characters ≥ 8 and ≤ 14	**No lower case character**	At least one upper case character	At least one numeric character	At least one character: ':', ';', '<', '=', '>', '?', '@'
8	Inputs outside all of the partitions above				

Figure 4: The table referenced in Exercise 2.

Table 5.2 Test design against predicate faults. BVA for predicate Age > 42

Program version no.	Correct/wrong predicate	Test data 1	Test data 2	Test data 3	Test data 4
	Age	Specific values of the variable Age			
		43 (ON)	42 (OFF)	20 (OUT)	50 (IN)
		Output			
1 (correct)	> 42	T	F	F	T
2	>= 42	T	T	F	T
3	< 42	F	F	T	F
4	<= 42	F	T	T	F
5	= 42	F	T	F	F
6	<> 42	T	F	T	T
7	> 43	F	F	F	T
8	> 41	T	T	F	T

Figure 5: The table referenced in Exercise 4.

Referenties

[1] A. K. István Forgács, *Practical Test Design: Selection of traditional and automated test design techniques.*