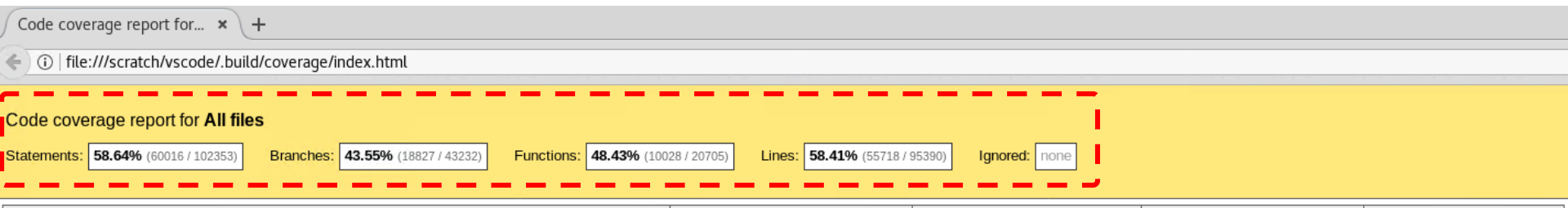


# Code Coverage Recap

# Example Code Coverage for Visual Studio Code



## Overall coverage

- Statements
- Branches
- Functions
- Lines

# Example Code Coverage for Visual Studio Code

Code coverage report for...					
file:///scratch/vscode/.build/coverage/index.html					
Code coverage report for All files					
Statements:	58.64% (60016 / 102353)	Branches:	43.55% (18827 / 43232)	Functions:	48.43% (10028 / 20705)
Lines:	58.41% (55718 / 95390)	Ignored:	none		
File	Statements	Branches	Functions	Lines	
out/vs/base/common/	32.84% (22 / 67)	20% (6 / 30)	23.08% (3 / 13)	33.33% (22 / 66)	
out/vs/base/test/common/	100% (3 / 3)	100% (0 / 0)	100% (2 / 2)	100% (2 / 2)	
src/vs/base/browser/	56.64% (1139 / 2011)	39.91% (362 / 907)	41.38% (156 / 377)	56.85% (1099 / 1933)	
src/vs/base/browser/ui/	46.67% (14 / 30)	100% (0 / 0)	11.11% (2 / 18)	54.55% (12 / 22)	
src/vs/base/browser/ui/actionbar/	24.82% (102 / 411)	6.86% (14 / 204)	12.64% (11 / 87)	23.65% (92 / 389)	
src/vs/base/browser/ui/aria/	31.25% (10 / 32)	16.67% (1 / 6)	60% (3 / 5)	31.25% (10 / 32)	
src/vs/base/browser/ui/button/	17.65% (21 / 119)	0% (0 / 51)	12% (2 / 15)	16.67% (19 / 114)	
src/vs/t					12 / 54
src/vs/t					8 / 107
src/vs/t					10 / 41
src/vs/t					3 / 129
src/vs/base/browser/ui/findinput/	24.64% (51 / 207)	0% (0 / 57)	9.3% (4 / 43)	22.16% (43 / 194)	
src/vs/base/browser/ui/highlightedlabel/	91.3% (42 / 46)	64.71% (11 / 17)	100% (6 / 6)	90.91% (40 / 44)	
src/vs/base/browser/ui/iconLabel/	26.74% (23 / 86)	0% (0 / 76)	19.05% (4 / 21)	25% (20 / 80)	
src/vs/base/browser/ui/inputbox/	16.13% (40 / 248)	1.85% (2 / 108)	4.55% (2 / 44)	15.97% (38 / 238)	
src/vs/base/browser/ui/keybindingLabel/	23.53% (12 / 51)	0% (0 / 48)	11.11% (1 / 9)	22.45% (11 / 49)	
src/vs/base/browser/ui/list/	26.72% (338 / 1265)	6.5% (26 / 400)	11.04% (37 / 335)	29.38% (322 / 1096)	
src/vs/base/browser/ui/menu/	33.33% (8 / 24)	0% (0 / 6)	28.57% (2 / 7)	31.82% (7 / 22)	
src/vs/base/browser/ui/octiconLabel/	71.43% (10 / 14)	0% (0 / 4)	42.86% (3 / 7)	69.23% (9 / 13)	
src/vs/base/browser/ui/progressbar/	79.61% (82 / 103)	48% (12 / 25)	63.64% (14 / 22)	79.59% (78 / 98)	
src/vs/base/browser/ui/sash/	43.62% (82 / 188)	25% (15 / 60)	36.36% (16 / 44)	45.35% (78 / 172)	
src/vs/base/browser/ui/scrollbar/	36.25% (232 / 640)	12.14% (34 / 280)	24.03% (31 / 129)	34.98% (212 / 606)	
src/vs/base/browser/ui/selectBox/	16.03% (76 / 474)	0% (0 / 196)	7.14% (7 / 98)	16.11% (72 / 447)	
src/vs/base/browser/ui/splitview/	49.9% (260 / 521)	37.63% (73 / 194)	37.9% (47 / 124)	49.89% (229 / 459)	
src/vs/base/browser/ui/toolbar/	28% (21 / 75)	0% (0 / 28)	13.04% (3 / 23)	27.54% (19 / 69)	
src/vs/base/common/	87.04% (5492 / 6310)	78.31% (2524 / 3223)	82.13% (933 / 1136)	87.33% (5219 / 5976)	
src/vs/base/common/diff/	98.83% (421 / 426)	94.42% (203 / 215)	100% (39 / 39)	98.81% (416 / 421)	

Identify the parts that are not being tested

# Beyond Functional Testing

- Code Coverage
- Equivalence Partitioning
- Boundary Value Analysis

# Equivalence Partitioning Motivation

- Test suites exercise a tiny fraction of our code's functionality

```
static int div(int a, int b)
{
    return a/b;
}
```

# Equivalence Partitioning Motivation

- Test suites exercise a tiny fraction of our code's functionality

```
static int div(int a, int b)
{
    return a/b;
}
```

- Exhaustive testing of all values ( $2^{32} \times 2^{32}$ ) is not possible!
- How do we choose a subset of values to test `div`?

# Equivalence Partitioning

- AKA *Equivalence Class*
- Equivalence Partitioning helps us “select a valuable fraction of the functionality most likely to be error-prone”

# Equivalence Partitioning

- Divides the input data into “partitions of equivalent data” separated by boundaries (see boundary value analysis)



# Equivalence Partitioning

- Divides the input data into “partitions of equivalent data” separated by boundaries (see boundary value analysis)

```
static int div(int a, int b) {...}
```

# Equivalence Partitioning

- Divides the input data into “partitions of equivalent data” separated by boundaries (see boundary value analysis)

$i..j$	$m..n$	$o..p$
--------	--------	--------

`static int div(int a, int b) {...}`

# Equivalence Partitioning

- Divides the input data into “partitions of equivalent data” separated by boundaries (see boundary value analysis)

Partition 1

$i..j$

Partition 2

$m..n$

Partition 3

$o..p$

`static int div(int a, int b) {...}`

# Equivalence Partitioning

- Divides the input data into “partitions of equivalent data” separated by boundaries (see boundary value analysis)

A partition represents  
a region of a method's parameter input space

Partition 1

Partition 2

Partition 3

$i..j$

$m..n$

$o..p$

`static int div(int a, int b) {...}`

# Equivalence Partitioning

The method behaves the same for all values within that partition/space/region

A partition represents a region of a method's parameter input space

Partition 1

Partition 2

Partition 3

$i..j$

$m..n$

$o..p$

`static int div(int a, int b) {...}`

# Example Equivalence Partitioning

- What are the equivalent partitions for parameter a?

```
static int div(int a, int b) {...}
```

# Example Equivalence Partitioning

- What are the equivalent partitions for parameter a?

<i>"negative"</i>	<i>"zero"</i>	<i>"pozitive"</i>
-------------------	---------------	-------------------

`static int div(int a, int b) {...}`

# Identifying Equivalence Partitions

- How many test cases will be needed to cover the regions in the following example?

```
static int div(int a, int b)
{
    return a/b;
}
```



# Identifying Equivalence Partitions

- How many test cases will be needed to cover the regions in the following example?

```
static int div(int a, int b)
{
    return a/b;
}
```

- **Nine** test cases:

- a can be *{positive, zero, negative}*
- b can be *{positive, zero, negative}*

# Equivalence Partitioning

- Is a **black-box technique**
- Helps generate test cases

# Beyond Functional Testing

- Code Coverage
- Equivalence Partitioning
- Boundary Value Analysis

# Boundary Value Analysis

- Black-Box Software testing technique
- Tests are designed to include values within specified boundaries
- Input conditions are divided into groups (classes)
  - input in the same class should behave similarly in the program
  - see Equivalence Partitioning

# Boundary Value Analysis Example 1

- Example from Monopoly game
- Test the feature/functionality “Go to Jail”
- Does the player have enough money to pay the \$50 fine?

# Boundary Value Analysis Example 1

- Example from Monopoly game
- Test the feature/functionality “Go to Jail”
- Does the player have enough money to pay the \$50 fine?

$< \$50$	$\geq \$50$
----------	-------------

# Boundary Value Analysis Example 2

- Potential (integer) input range:  $a \dots b$
- Test with values:

# Boundary Value Analysis Example 2

- Potential (integer) input range:  $a \dots b$
- Test with values:
  - $a - 1$
  - $a$
  - $a + 1$
  - some value between  $a$  and  $b$  (for equivalence partition)
  - $b - 1$
  - $b$
  - $b + 1$



# Boundary Value Analysis Example 2

- Potential (integer)\* input range:  $a \dots b$
- Test with values:
  - $a - 1$
  - $a$
  - $a + 1$
  - some value between  $a$  and  $b$  (for equivalence partition)
  - $b - 1$
  - $b$
  - $b + 1$

\*For non-integer range, test values slightly less  $a$  and slightly more than  $b$

# Boundary Value Analysis Example 3

- Months of the year expressed as integers
- Equivalent partitions are:

# Boundary Value Analysis Example 3

- Months of the year expressed as integers
- Equivalent partitions are:
  - any input from the invalid partition should fail
  - any input from the valid partition should pass

..., -2, -1, 0	1 ... 12	13, 14, ...
Invalid partition	Valid partition	Invalid partition

# Other Types of Test Cases to Consider

- Can something cause **division by zero**?
- What if the **input type is wrong**
  - expecting an integer, but a float is given
  - expecting a character, but an integer is given
- What if **resources** (i.e., file handlers) **are left open** after program terminates?

These types of errors can be easily detected using  
**static analysis** tools

# Other Types of Test Cases to Consider

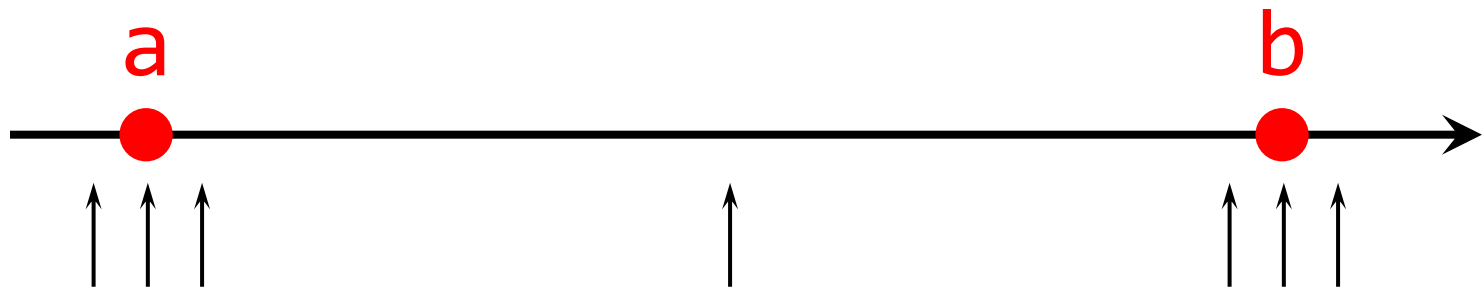
- What if **mandatory fields** are not entered?
- What if the program is **aborted abruptly** or input or output devices are unplugged?
- What if the **customer takes an “illogical” path** (i.e., unexpected path) through your program?

# Equivalence Partitioning and Boundary Value Analysis Summary

- Defects really do tend to arise at region boundaries
- So many of our tests exercised not just one case in a region but several around the boundaries between partitions

# Boundary Value Analysis Summary

- Given a range **a...b** we would like to test:
  - around **a**
  - some middle value (equivalence partition)
  - around **b**
- Focus is towards the boundaries



# Summary

- NB: Neither *Structural Testing* (*Code Coverage*) and *Equivalence Partitioning / Value Boundary Analysis* are adequate by themselves
  - High-quality products use both
- When to stop testing?



# Summary

- NB: Neither *Structural Testing* (*Code Coverage*) and *Equivalence Partitioning / Value Boundary Analysis* are adequate by themselves
  - High-quality products use both
- When to stop testing?
  - “Stop testing when fear turns into boredom”
  - Is still an open-ended question

# Bibliography

- [1992Grady] Grady, Robert. Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall. 1992.
- [2012Fowler]: <http://martinfowler.com/bliki/TestCoverage.html>

# Whiteboard Only Exercise

# What Type of Testing is This?

- See video of running Visual Studio Code tests at:

<https://drive.google.com/file/d/1CBKD-6iQp91UiYLDW2kujFnW5tWoZfbO/view?usp=sharing>

- The tests in the videos were generated using the following command:

```
yarn smoketest
```

- Additional Resources:

- <https://github.com/Microsoft/vscode/wiki/How-to-Contribute#build-and-run-from-source>
- <https://github.com/Microsoft/vscode/blob/master/test/smoke/README.md>