



CS304 SOFTWARE ENGINEERING

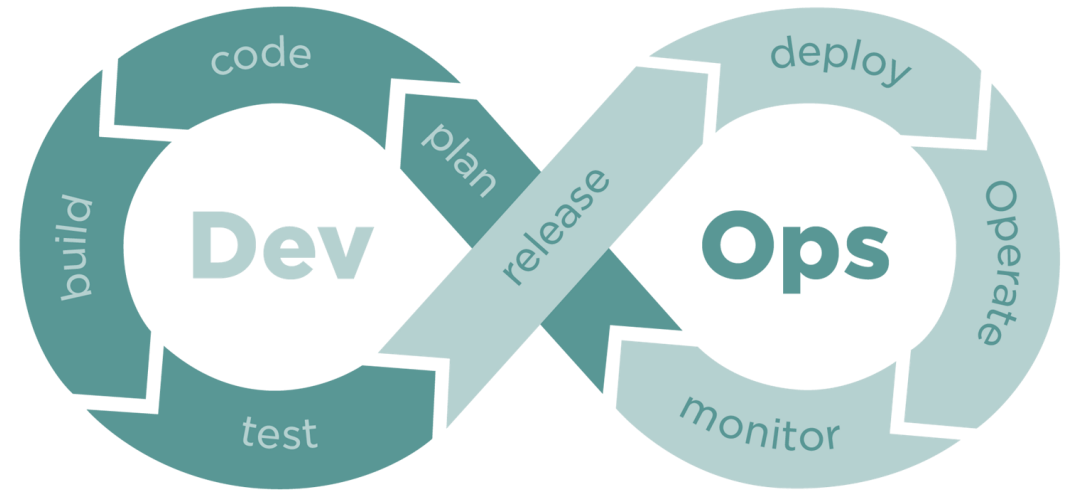
Yida Tao

taoyd@sustech.edu.cn

WHERE ARE WE NOW?

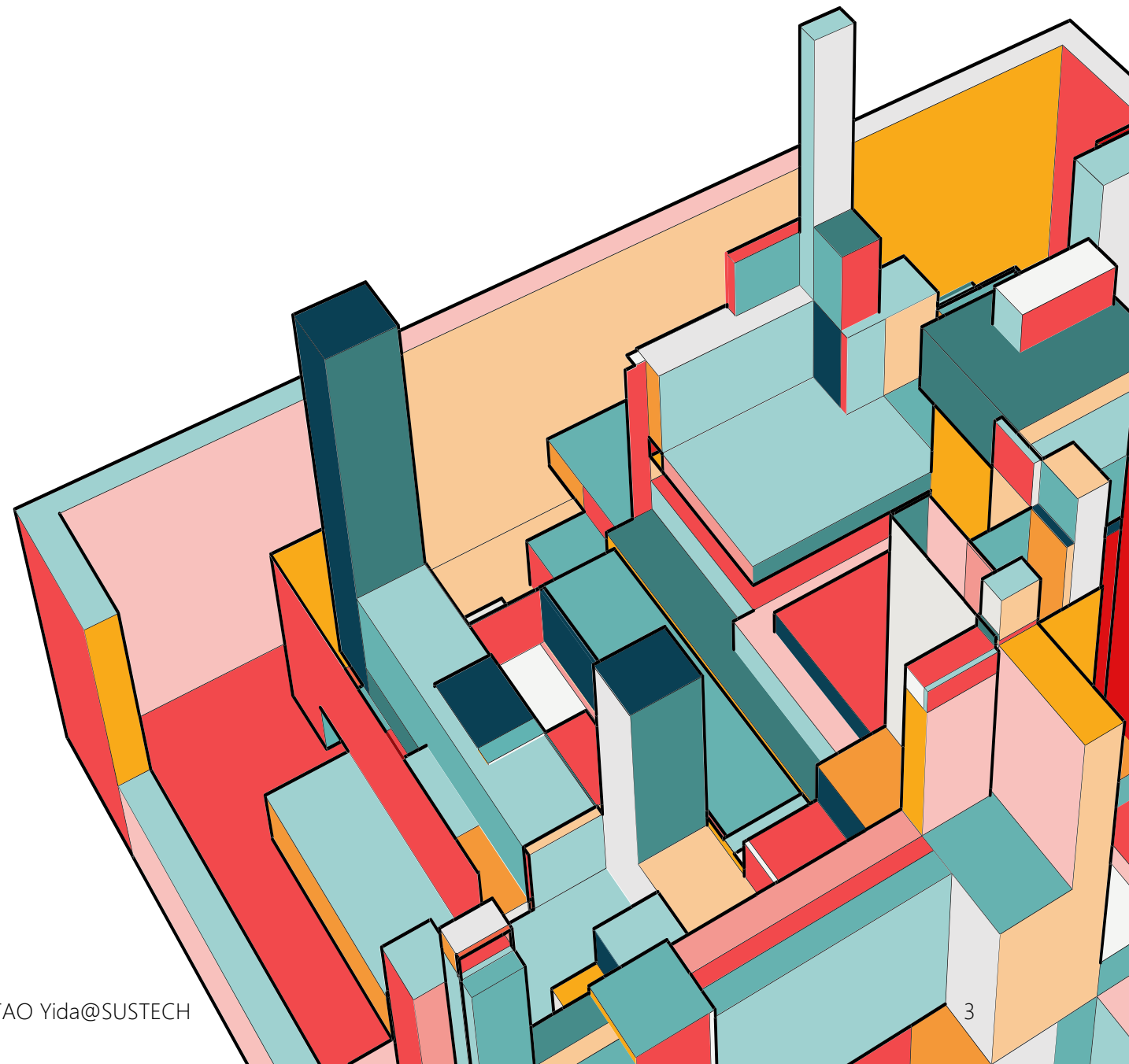
Quality control

- Code quality
- **Testing**



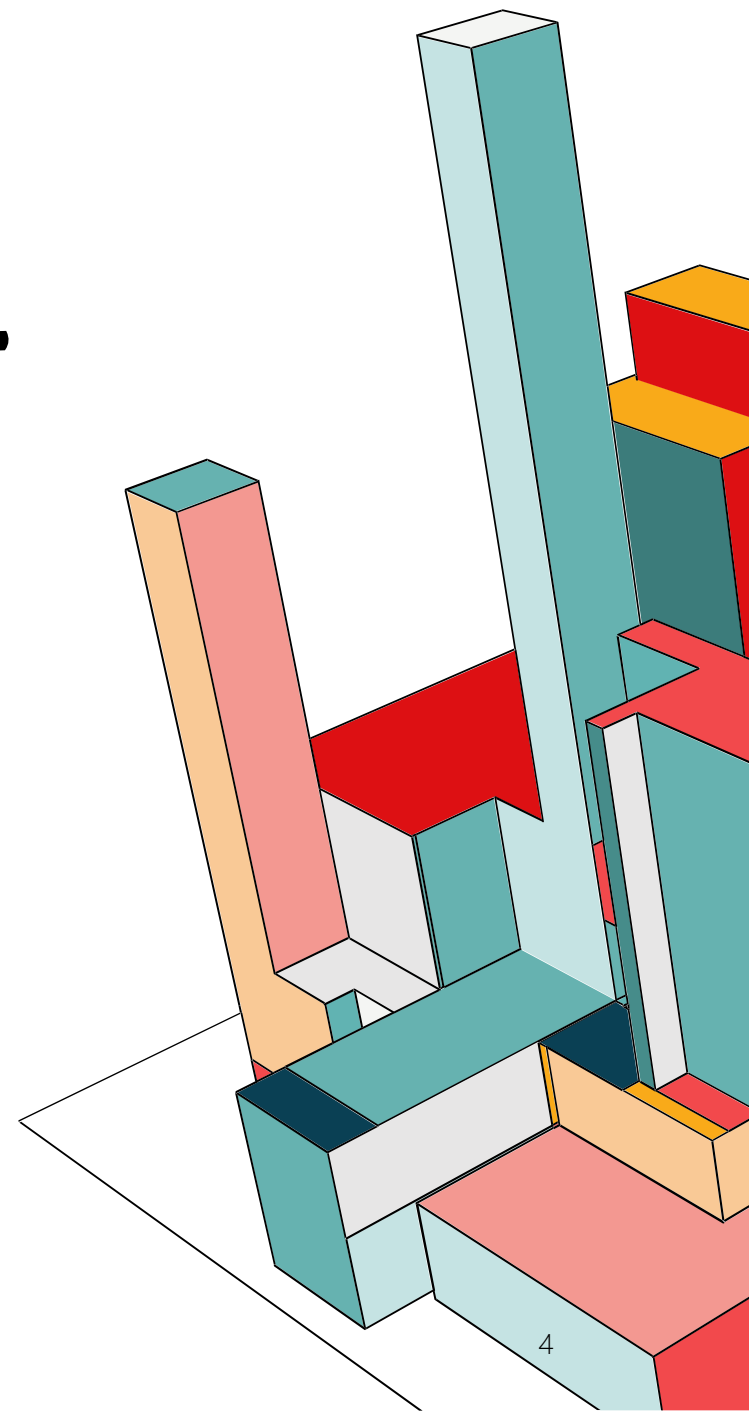
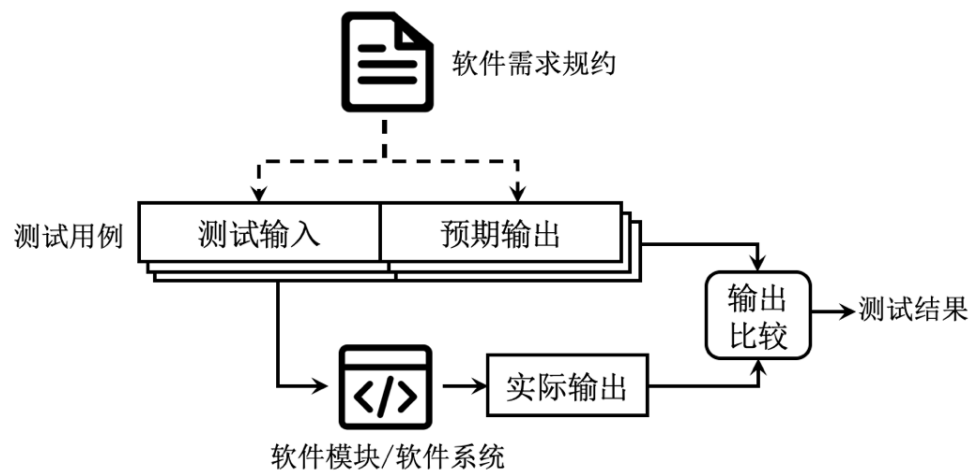
LECTURE 8

- Overview
- Important Concepts
- Blackbox & Whitebox Testing



SOFTWARE TESTING IN A NUTSHELL

- We got a software
- We throw required input at it
- We check whether the software performed as we expected



HISTORY OF TESTING



Why bother testing?

Largely manual and error prone

Developer-driven, automated testing

Image source: <http://cartoontester.blogspot.com/2010/11/my-first-day-at-expoqa.html>

EARLY TESTING PRACTICE @ GOOGLE

- In Google's early days, engineer-driven testing was often assumed to be of little importance.
- Teams regularly relied on smart people to get the software right.
- A few systems ran large integration tests, but mostly it was **the Wild West**.

One product that suffers the most is **Google Web Server (GWS)**, which is the web server responsible for serving Google Search queries

THE STORY OF GOOGLE WEB SERVER (GWS)

- Back in 2005, as the project swelled in size and complexity, productivity had slowed dramatically.
- Releases were becoming buggier, and it was taking longer and longer to push them out.
- Team members had little confidence when making changes to the service, and often found out something was wrong only when features stopped working **in production**.

At one point, more than 80% of production pushes contained user-affecting bugs that had to be rolled back.

THE STORY OF GOOGLE WEB SERVER (GWS)

To address these problems, the tech lead of GWS decided to institute a policy of

engineer/developer-driven, automated testing

DEVELOPER DRIVEN TESTING

Every coded feature should be tested completely once, by the developer responsible for implementing it

- In Developer Driven Testing (DDT), developers write their tests by themselves, only when they feel they're done with the code.
- They are the people who have to verify that the code is “good enough,” not the testers.
- The purpose of this approach is to make developers responsible for their code.

At Google, all new code changes were required to include tests

AUTOMATED TESTING

Another key component for CI/CD!

Manual testing won't scale for modern software development

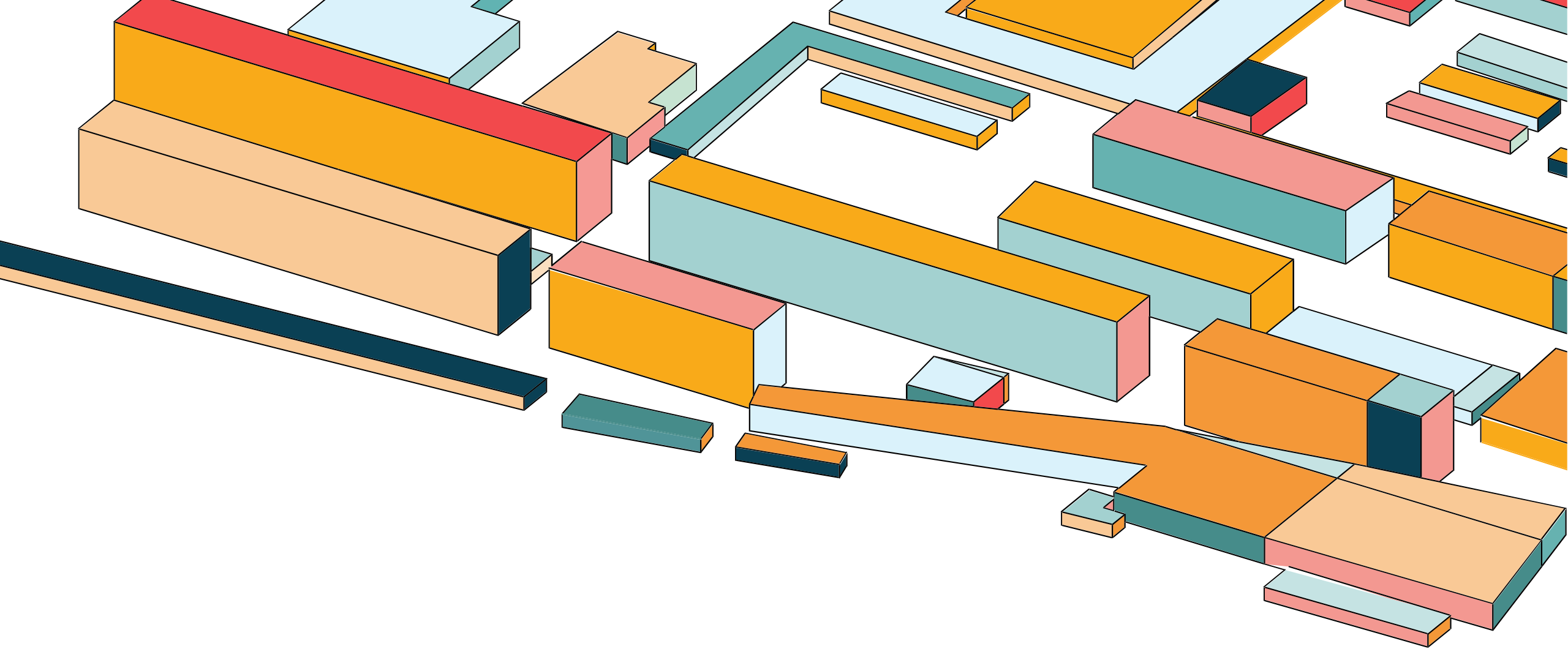
- Thousands of lines of code
- Hundreds of libraries and frameworks
- Multiple platforms & languages
- Uncountable number of configurations
- Frequent updates or new versions
- Delivered via unreliable networks

At Google, all new code changes were required to include tests, which run automatically and continuously (e.g., thousands of times per day)

OUTCOME

- Within a year of instituting this policy, the number of emergency pushes *dropped by half*.
- This drop occurred despite the fact that the project was seeing a record number of new changes every quarter.
- Even in the face of unprecedented growth and change, **testing brought renewed productivity and confidence** to one of the most critical projects at Google.

Today, GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures.



IMPORTANT CONCEPTS IN SOFTWARE TESTING

TEST CASE VS. TEST SUITE

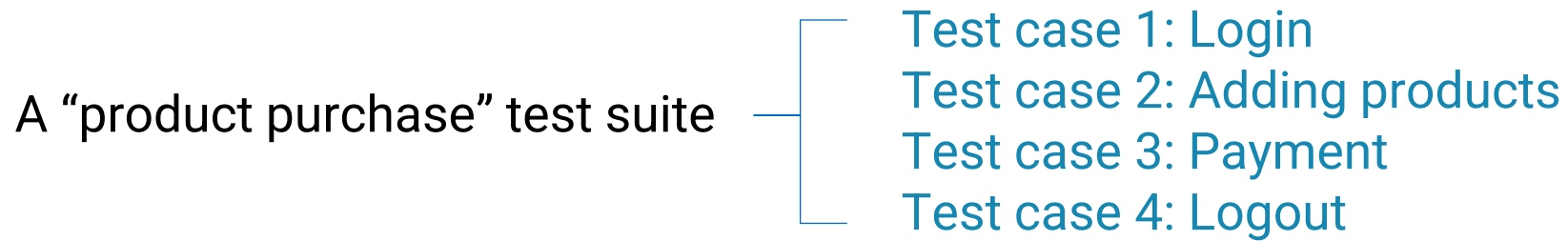
- **Test case** (测试用例): a sequence of actions necessary to verify a specific functionality or feature of the software.
- A test case specifies the prerequisites, post conditions, steps, and data required for feature verification.

Test case example: Check results when a valid Login Id and Password are entered.

<https://www.ibm.com/docs/en/elm/7.0.3?topic=scripts-test-cases-test-suites>

TEST CASE VS. TEST SUITE

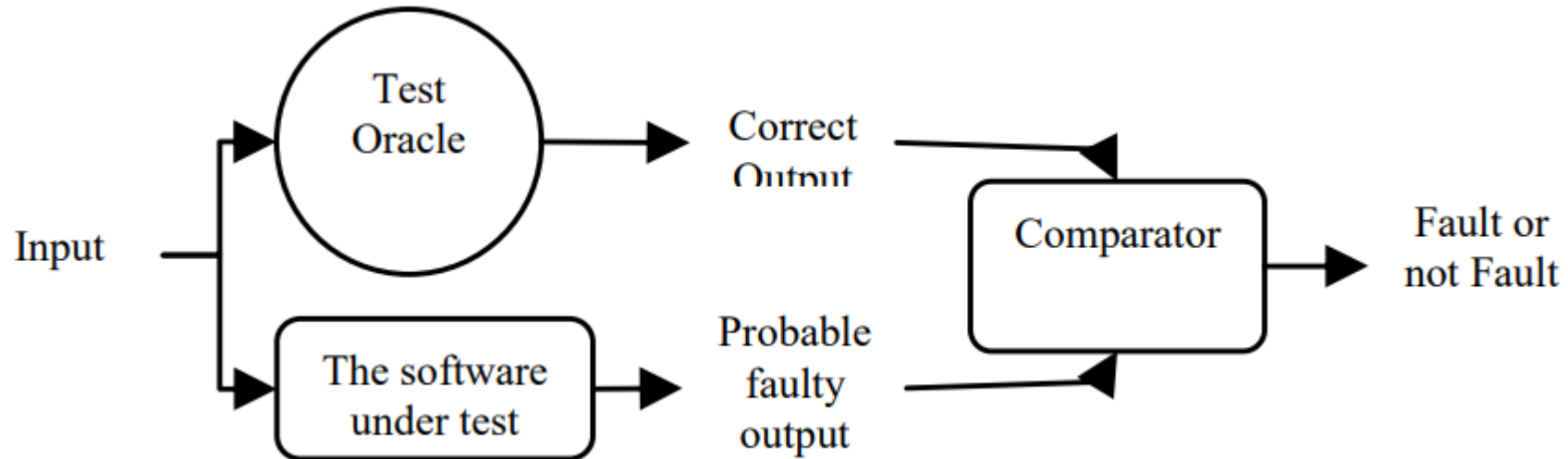
- **Test suites** (测试套件、测试集) are the logical grouping or collection of test cases to run a single job with different test scenarios.
- Example: a test suite for product purchase has multiple test cases



<https://www.ibm.com/docs/en/elm/7.0.3?topic=scripts-test-cases-test-suites>

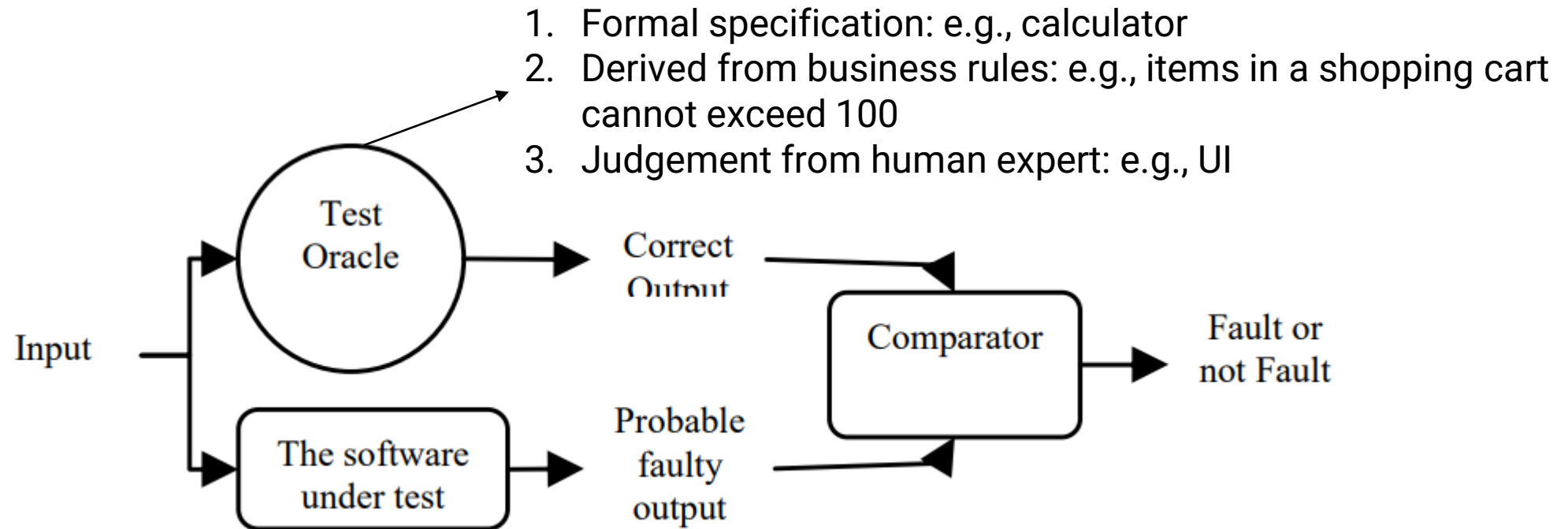
TEST INPUT VS. TEST ORACLE

Test Oracle (测试预言、测试判断准则) is a fault free source of expected outputs: it accepts every test input specified in software's specification and should always generate a correct result



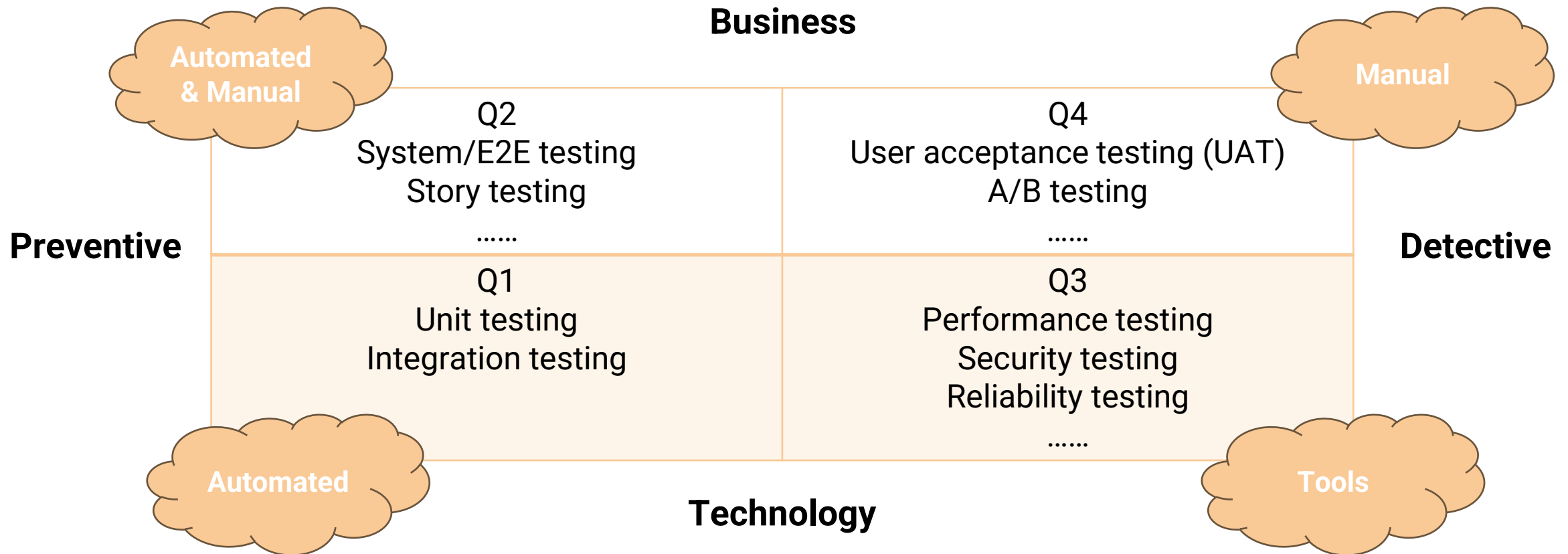
Kadir, W.M., & Nasir, W.M. (2008). Intelligent and automated software testing methods classification.

TEST INPUT VS. TEST ORACLE



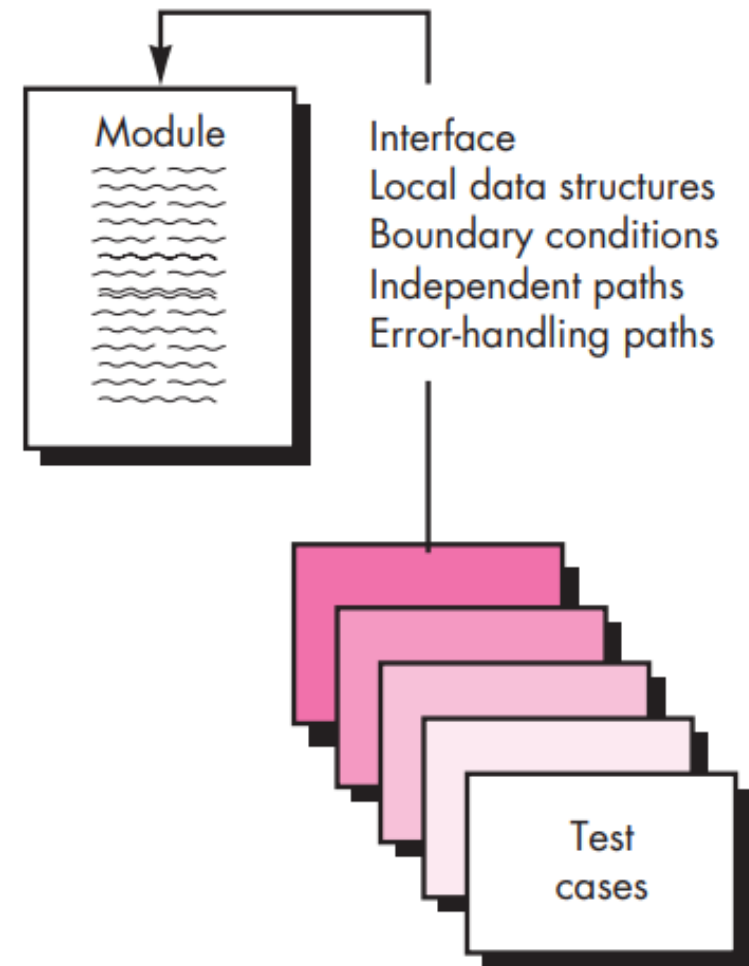
Kadir, W.M., & Nasir, W.M. (2008). Intelligent and automated software testing methods classification.

TESTING QUADRANTS



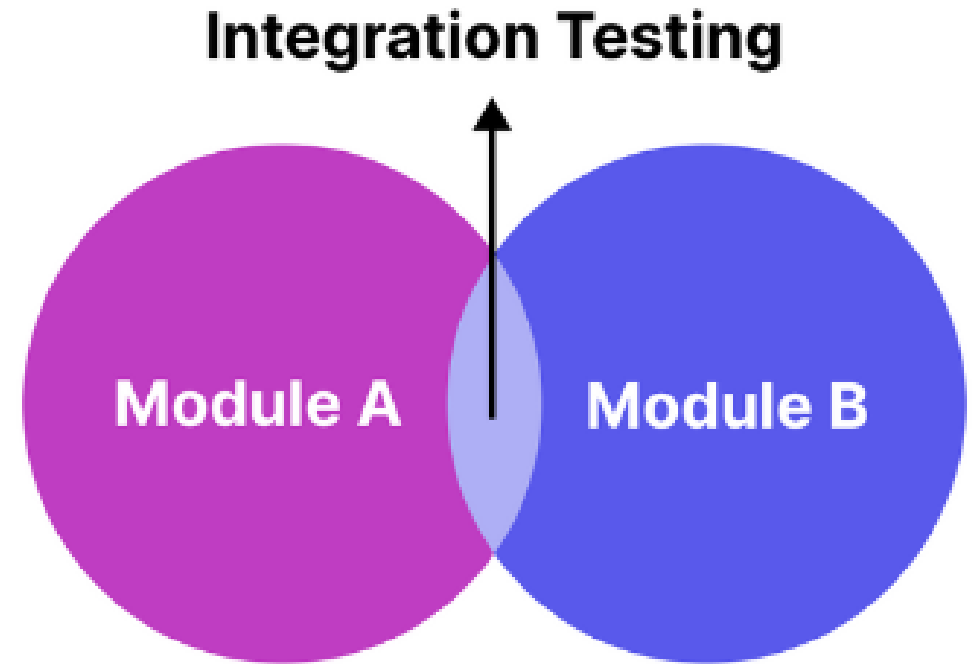
UNIT TESTING

- Unit testing focuses verification effort on the smallest unit of software design, e.g., **methods, classes**
- Unit tests focus on the internal processing logic and data structures within the boundaries of a component.
- By isolating each unit and testing it independently, unit testing can be conducted in parallel for multiple components.



INTEGRATION TESTING

- Unit testing ensures that components work individually
- Integration testing verifies the interactions and behavior of multiple components/modules working together (through interfaces)



<https://katalon.com/resources-center/blog/unit-testing-integration-testing>

INTEGRATION TESTING

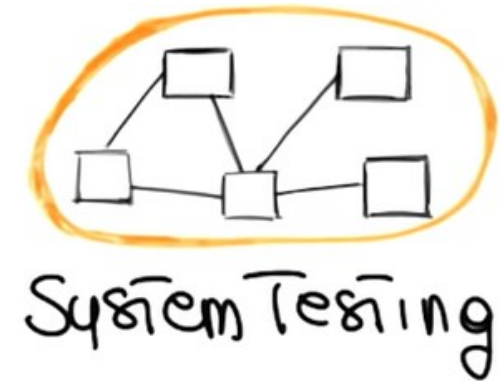
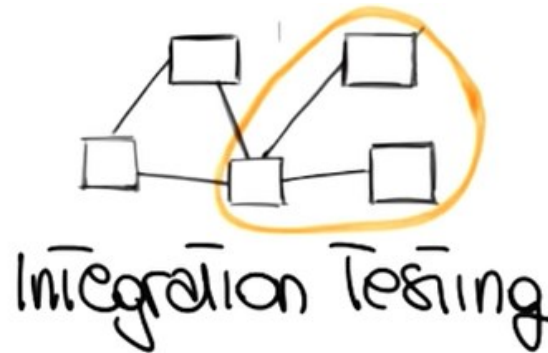
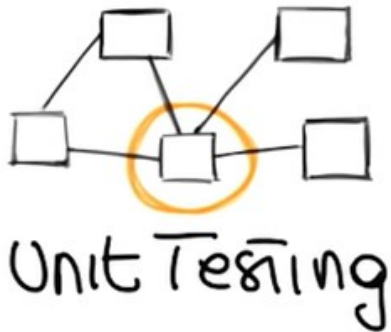
Test Case ID	Test Case Objective	Test Case Description	Expected Result
1	Check the interface link between the Login and Mailbox module	Enter login credentials and click on the Login button	To be directed to the Mail Box
2	Check the interface link between the Mailbox and Delete Mails Module	From Mailbox select the email and click a delete button	Selected email should appear in the Deleted/Trash folder

An example of integrating testing involving a website that features “Log-in Page,” “Mailbox,” and “Delete E-mails” functions.

<https://www.guru99.com/integration-testing.html>

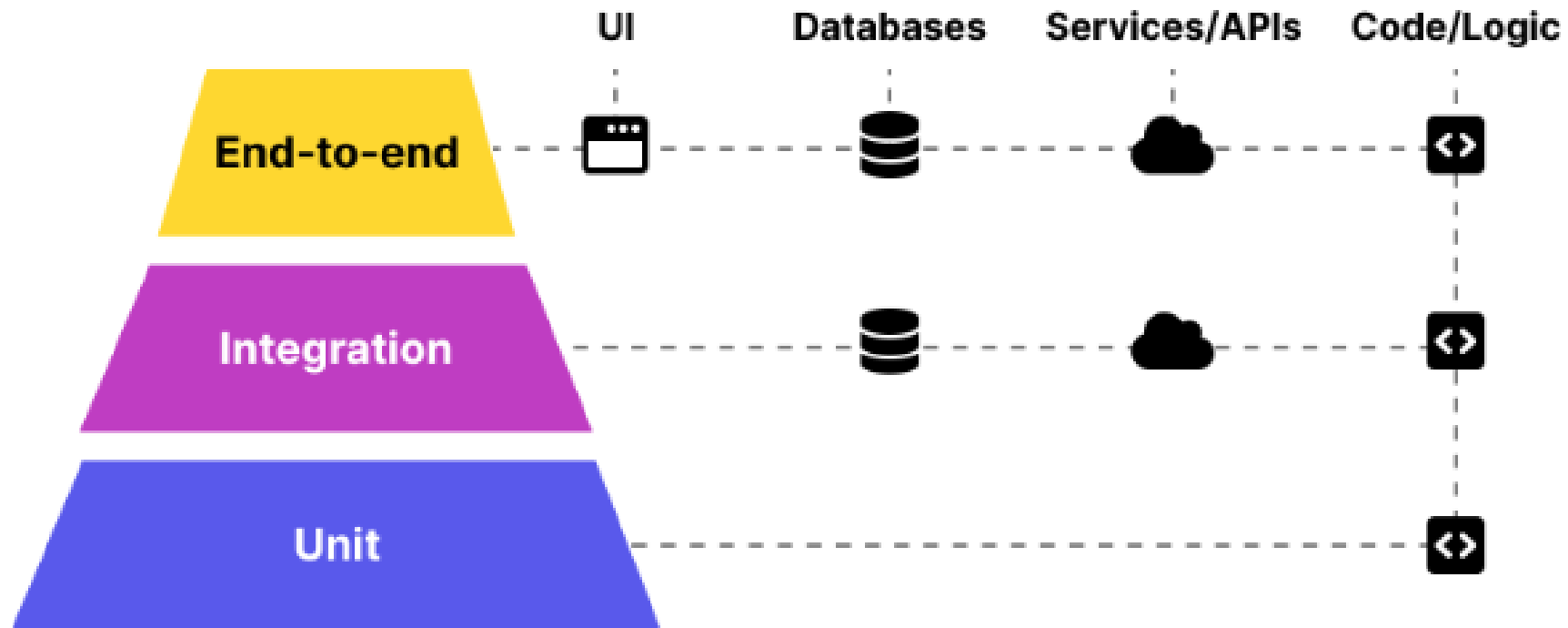
SYSTEM TESTING

- System testing aims at testing that the product fulfills the business specifications and is ready for deployment
- System testing is generally executed after the integration test and before the delivery of the software product
- System testing is often conducted in an environment closely resemble the production environment



<https://youtu.be/SjVfvXJeaJA>

A COMPARISON

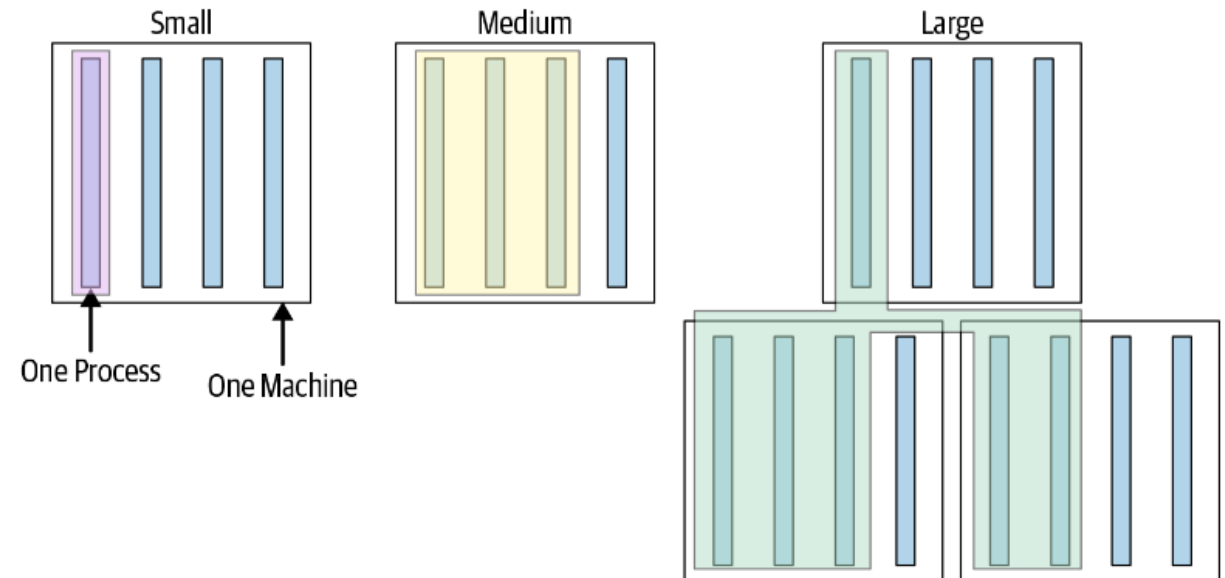


TEST SIZE

How much resources are consumed by a test

Small tests

- run in a single thread/process, no blocking calls (e.g., no network calls)
- provide a safe “sandbox”
- fast, effective, and reliable

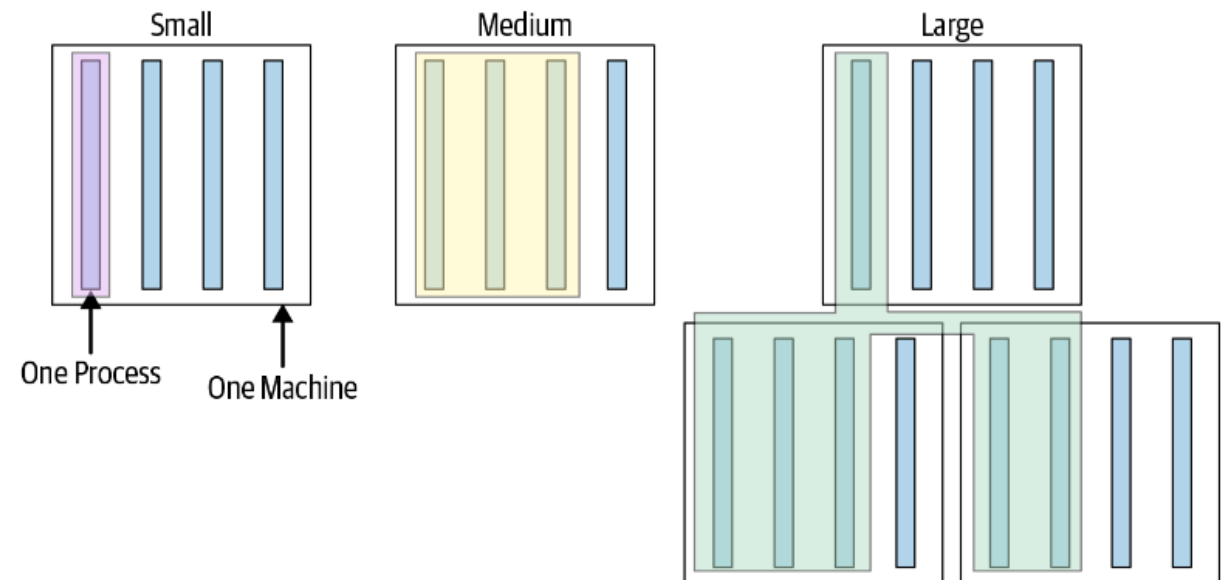


TEST SIZE

How much resources are consumed by a test

Medium tests

- Run on a single machine
- can span multiple processes, use threads, and can make blocking calls (e.g., network calls **only to localhost, but not to remote machines via network**)
- Enable testing the integration of multiple components (database, UI, server, etc.)

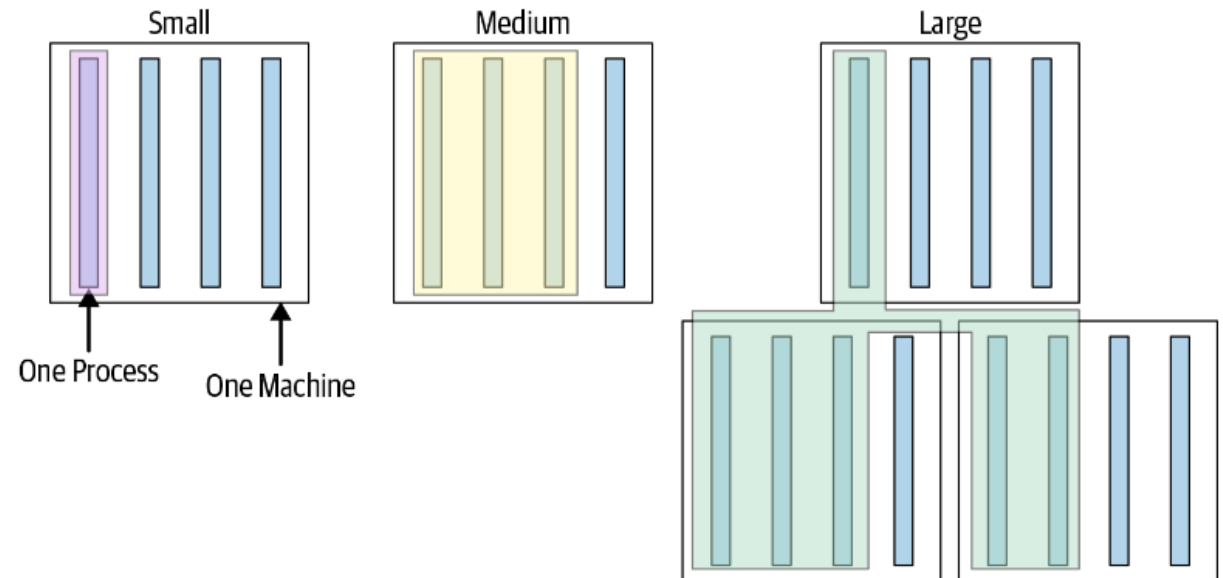


TEST SIZE

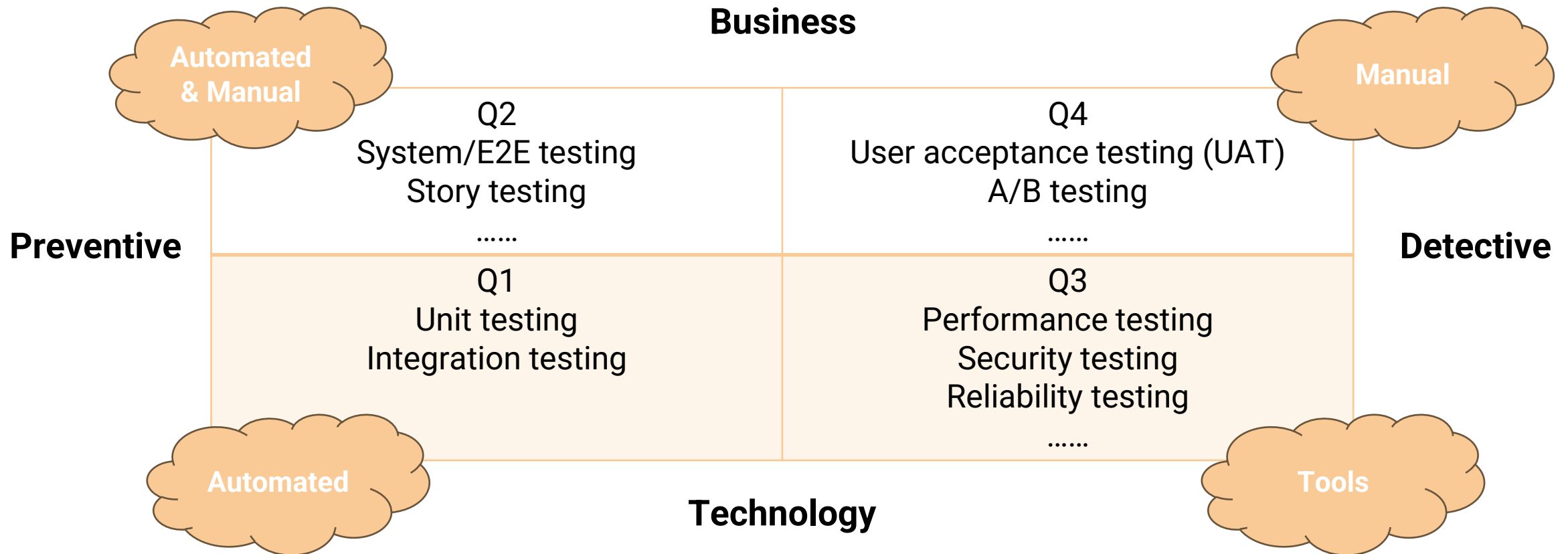
How much resources are consumed by a test

Large tests

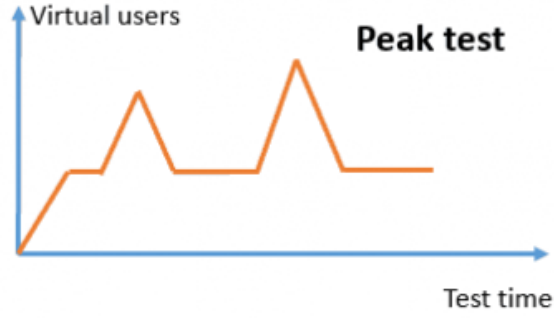
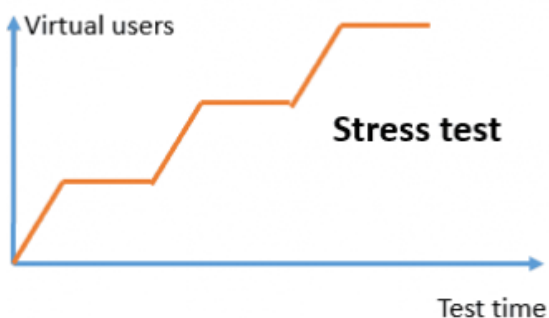
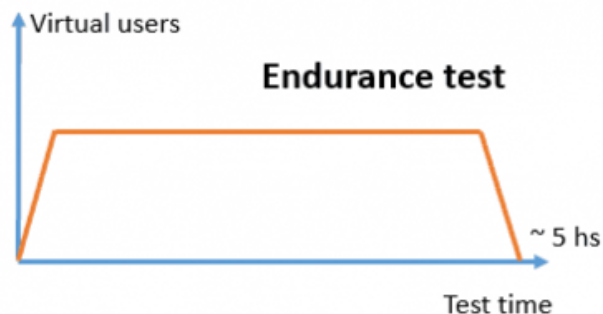
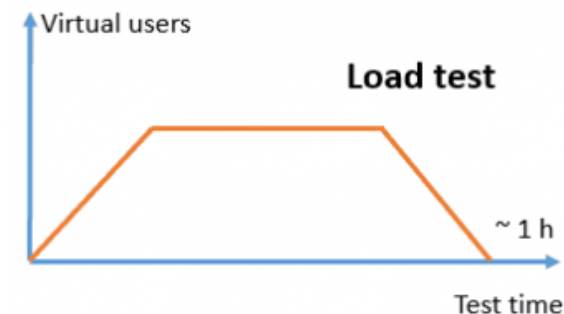
- Run on a multiple machines over network
- More about validating configurations instead of piece of code
- Reserved for full system end-to-end tests



MOVING ON TO Q3 AND Q4



PERFORMANCE TESTING



Load testing

Load testing is a type of testing which involves evaluating the performance of the system under the expected workload.

Stress testing

Stress testing is a type of performance testing where we evaluate the application's performance at load much higher than the expected load.



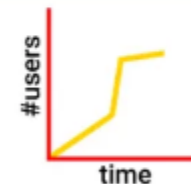
Endurance testing

Endurance testing is also known as 'Soak Testing'. It is done to determine if the system can sustain the continuous expected load for a long duration.



Spike testing

In spike testing, we analyze the behavior of the system on suddenly increasing the number of users.



Volume testing

The volume testing is performed by feeding the application with a high volume of data.

<https://artoftesting.com/types-of-performance-testing>

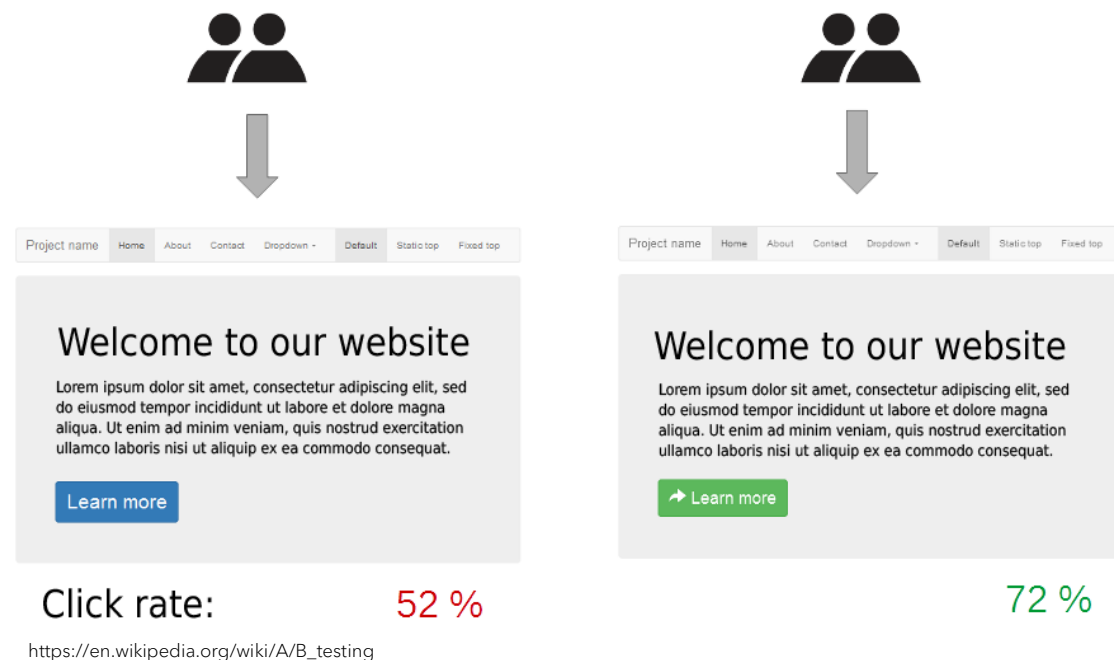
USER ACCEPTANCE TESTING (UAT)

- The last phase of software testing before making the system available for actual use.
- UAT is used to determine whether the product is working for the end-users correctly
- UAT is performed by **business users** to verify that the application will meet the needs of the **end-user**, with scenarios and data representative of **actual usage** in the field.

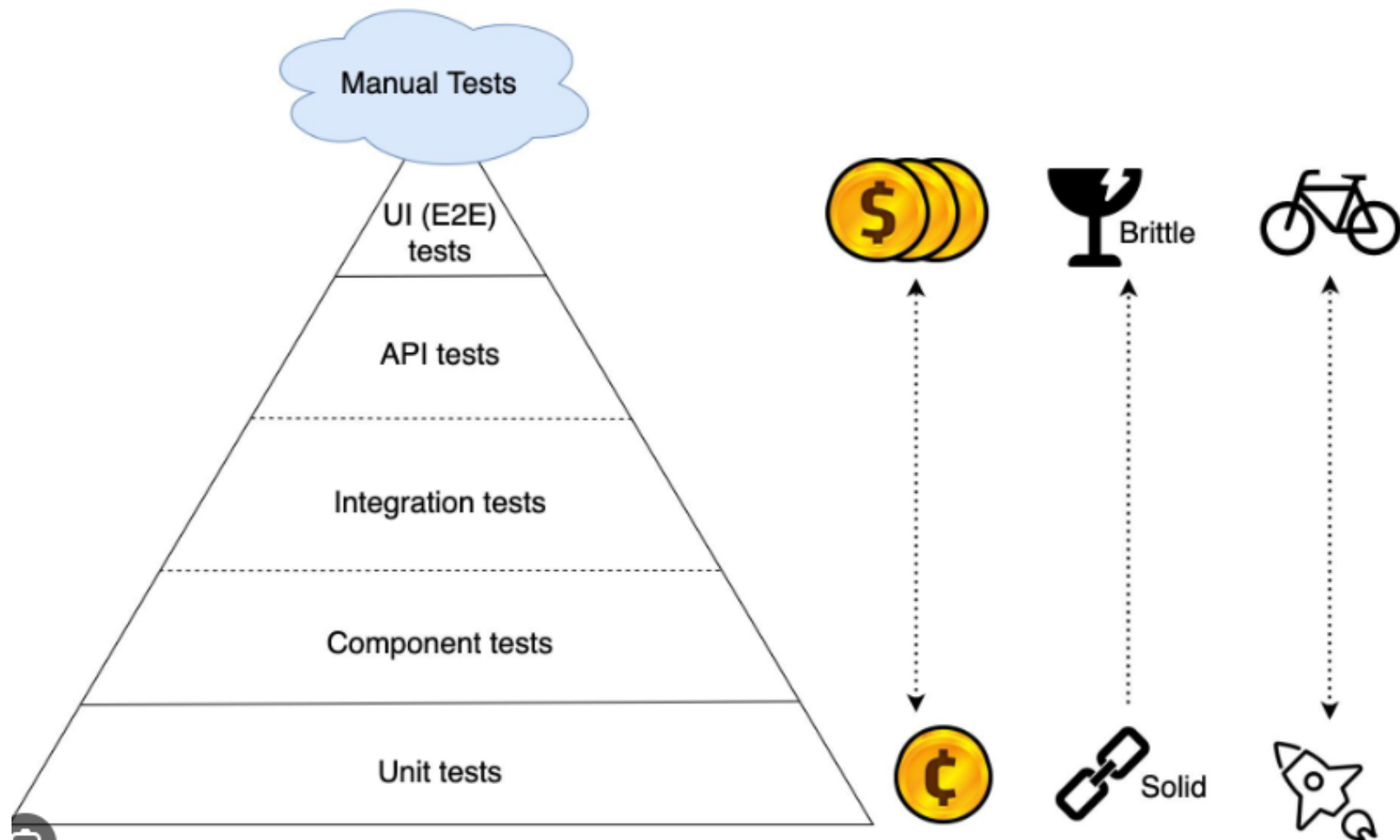


A/B TESTING

- A/B testing (also known as bucket testing, split-run testing, or split testing) is a user experience research methodology
- A/B testing compares multiple versions of a single variable (e.g., **the color of a button**), and determines which of the variants is more effective.

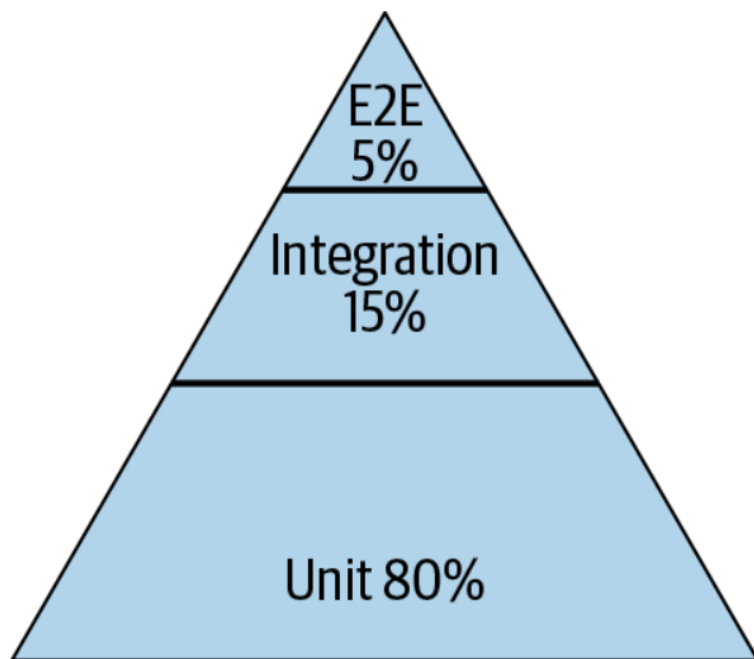


TESTING PYRAMID



<https://getmason.io/blog/post/test-pyramid/>

TESTING PYRAMID



Google's test pyramid

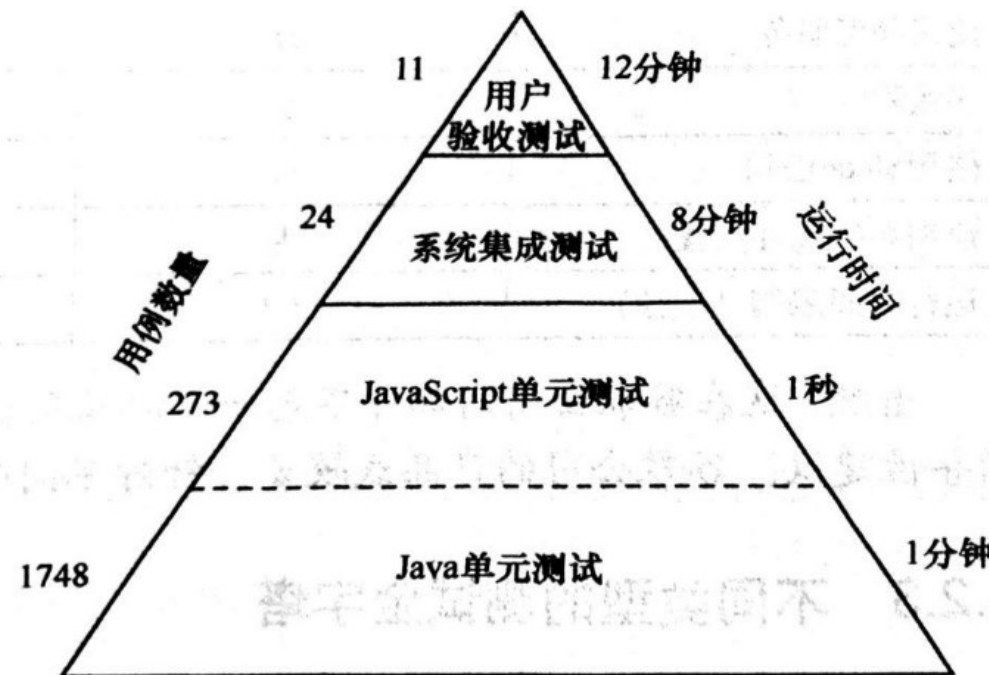
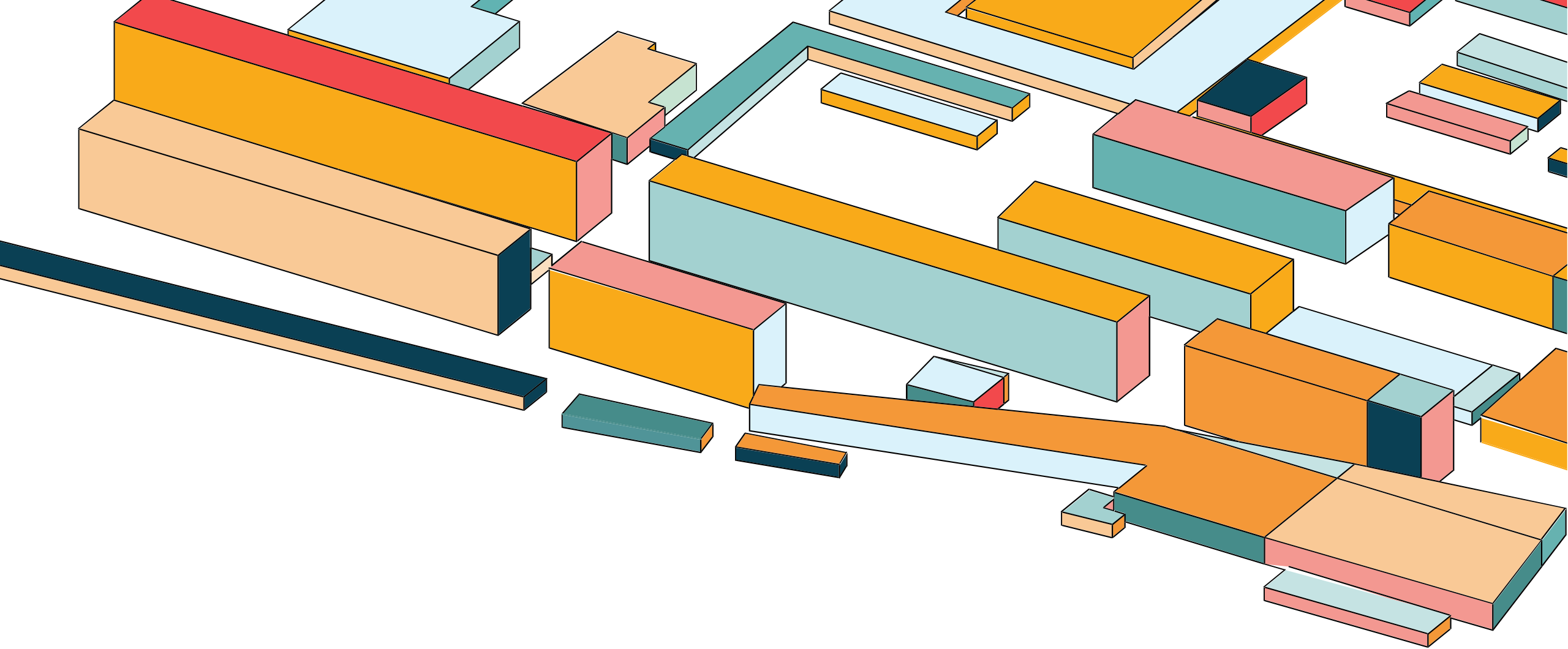


图10-6 某J2EE软件应用的自动化测试数量与运行时间

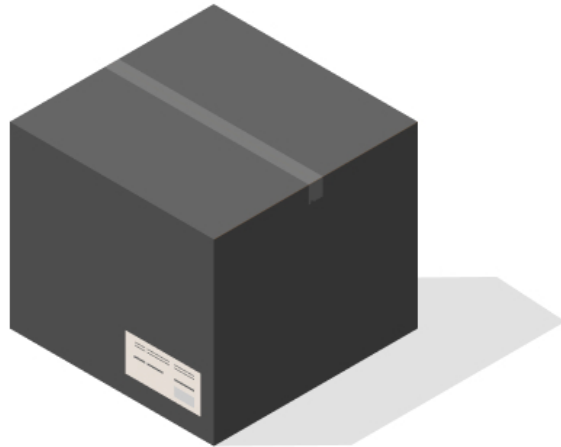
Image source: 《持续交付2.0》乔梁



TESTING TECHNIQUES

Part of the examples are based on
https://youtube.com/playlist?list=PLAwTw4SYaPkoQFThzsc9e7Fe3QV_KJCs

BLACK-BOX TESTING VS. WHITE-BOX TESTING



Black-box testing

- Based on software specification
- Internal program structure is unknown
- Cover as much specified behaviors as possible

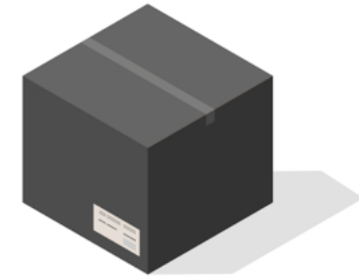


White-box testing

- Based on software code
- Internal program structure is known
- Cover as much coded behaviors as possible

EXAMPLE I

Specification: inputs an integer and prints it



Test possible integers based on specification,
e.g., 1, 0, -1

Miss the bug

EXAMPLE I

Specification: inputs an integer and prints it

Possible implementation

```
void print(num){  
    if(param<1024) printf("%d", num);  
    else printf("%d KB", num/124);  
}
```



Test every possible code path, e.g., 1000, 2000

Detect the bug

EXAMPLE II

Possible implementation

```
int func(int num){  
    int result;  
    result = num/2;  
    return result;  
}
```



Test every possible code path, e.g., 2, 200

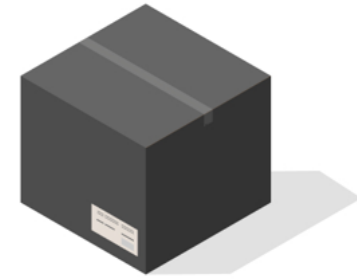
Miss the bug

EXAMPLE II

Specification: inputs an integer and returns half of its value for even integers, and the same value otherwise

Possible implementation

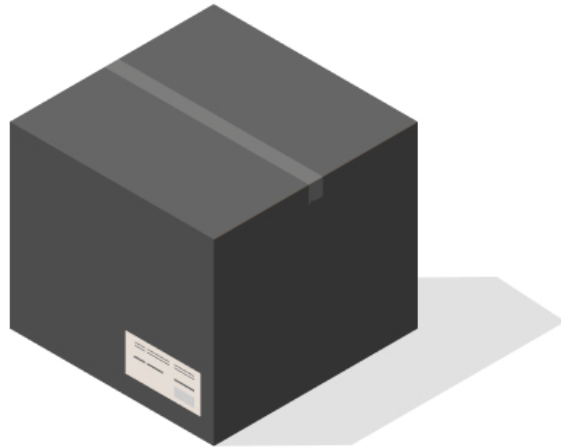
```
int func(int num){  
    int result;  
    result = num/2;  
    return result;  
}
```



Test every behavior in the specification,
e.g., 2, 3

Detect the bug

BLACK-BOX TESTING VS. WHITE-BOX TESTING



Black-box testing

- Pros: simplicity, realistic results (simulate end users)
- Cons: limited coverage, incomplete testing



White-box testing

- Pros: comprehensive testing, early defect detection
- Cons: requires technical expertise, expensive, limited real-world simulation.

WHITE-BOX TESTING

- One of the main goals of white box testing is to cover the source code as comprehensively as possible.
- Assumption: executing the faulty statement is a necessary condition for revealing a fault
- Code coverage is a metric that shows how much of an application's code has tests checking its functionality

WHITE-BOX TESTING - STATEMENT COVERAGE

- Statement coverage aims at executing all possible statements of the source code at least once.
- It is used to calculate the total number of executed statements in the source code out of total statements present in the source code.

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

WHITE-BOX TESTING - STATEMENT COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

Test case # 1
a = 1, b = 2

Statement coverage = $5/7 \approx 71\%$

WHITE-BOX TESTING - STATEMENT COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

Test case # 2
a = -2, b = -4

Statement coverage = $6/7 \approx 85\%$

WHITE-BOX TESTING - STATEMENT COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

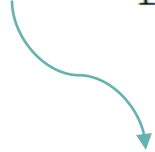
Test case # 1
a = 1, b = 2

Test case # 2
a = -2, b = -4

Statement coverage = 100%

WHITE-BOX TESTING - STATEMENT COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```



Test case # 1
a = 1, b = 2

Test case # 2
a = -2, b = -4

What if the method should also print “zero” when result == 0?
Even 100% statement coverage won’t reveal this fault

WHITE-BOX TESTING - BRANCH COVERAGE

- A "branch" is one of the possible execution paths the code can take after a decision statement—e.g., an if statement—gets evaluated.
- Branch coverage is a requirement that, for each branch in the program, each branch have been executed at least once during testing (i.e., each branch condition must have been TRUE at least once and FALSE at least once)

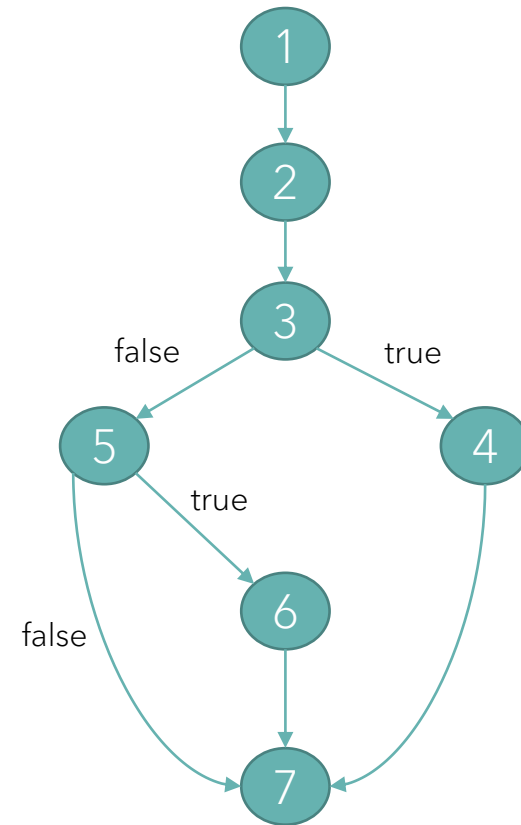
Branch coverage = (# of executed branches / # of total branches) x 100%

WHITE-BOX TESTING - BRANCH COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

Test case # 1
a = 1, b = 2

Branch coverage:
 $1 / 4 = 25\%$

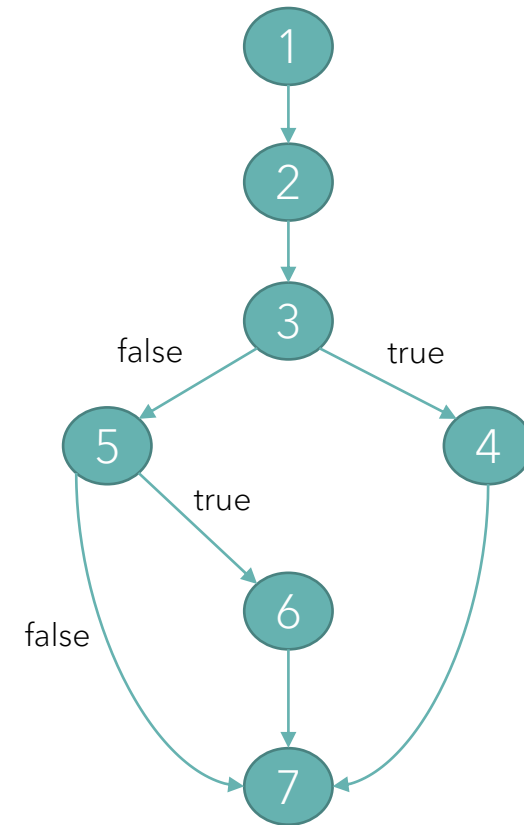


WHITE-BOX TESTING - BRANCH COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

Test case # 2
a = -2, b = -4

Branch coverage:
 $2 / 4 = 50\%$



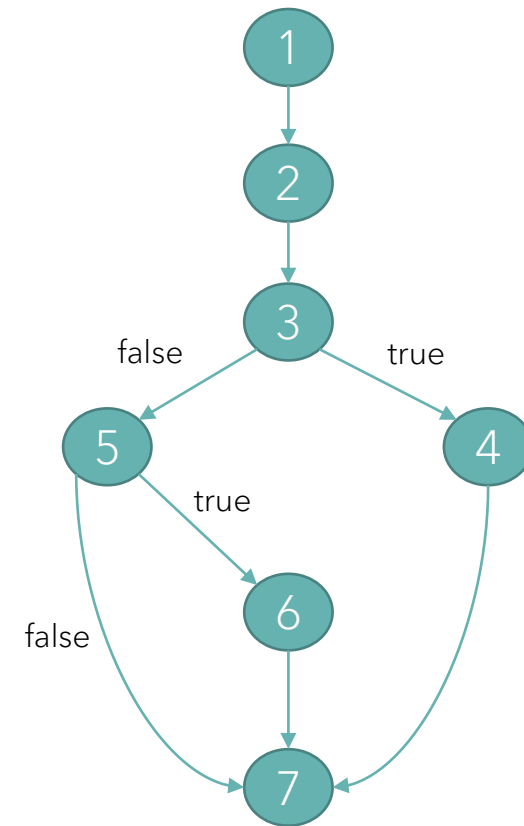
WHITE-BOX TESTING - BRANCH COVERAGE

```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

Test case # 1
a = 1, b = 2

Test case # 2
a = -2, b = -4

Branch coverage:
 $3 / 4 = 75\%$



WHITE-BOX TESTING - BRANCH COVERAGE

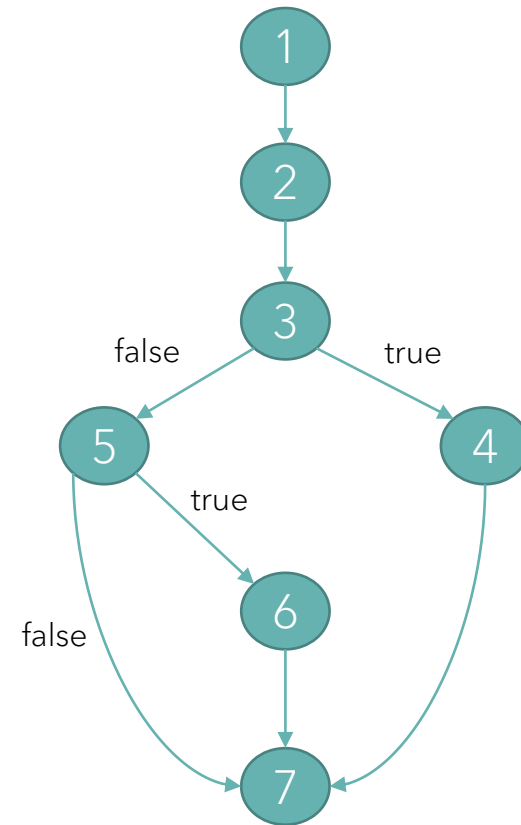
```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

Test case # 1
a = 1, b = 2

Test case # 2
a = -2, b = -4

Test case # 3
a = 0, b = 0

Branch coverage:
4 / 4 = 100%



WHITE-BOX TESTING - BRANCH COVERAGE

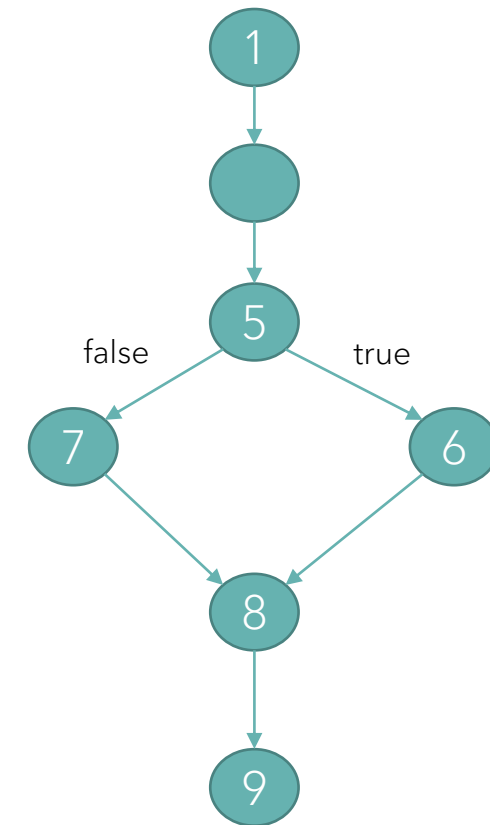
```
1 func() {  
2     int x, y;  
3     read(x);  
4     read(y)  
5     if( x == 0 || y > 0)  
6         y = y/x;  
7     else x = y + 2;  
8     write(x);  
9     write(y)  
10 }
```

Yet we still miss this
division-by-zero error!

Test case # 1
x = 1, y = 2

Test case # 2
x = 1, y = -2

Branch coverage: $2 / 2 = 100\%$



WHITE-BOX TESTING - CONDITION COVERAGE

- Condition coverage is also known as Predicate Coverage, in which each one of the boolean expression (condition) should have been evaluated to both TRUE and FALSE.

Condition coverage = (# of conditions that are both T and F / # total conditions) x 100%

WHITE-BOX TESTING - CONDITION COVERAGE

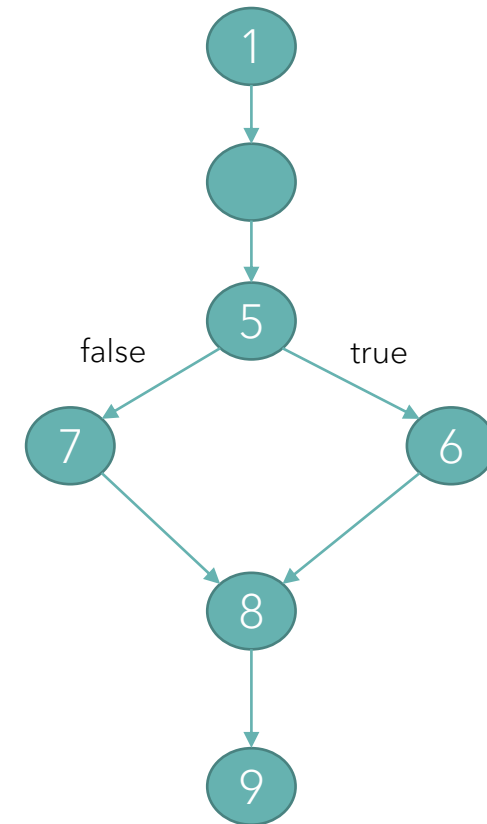
```
1 func() {  
2     int x, y;  
3     read(x);  
4     read(y)  
5     if( x == 0 || y > 0)  
6         y = y/x;  
7     else x = y + 2;  
8     write(x);  
9     write(y)  
10 }
```

Yet we still miss
the else branch!

Test case # 1
 $x = 0, y = -5$

Test case # 2
 $x = 5, y = 5$

Condition coverage: $2 / 2 = 100\%$



WHITE-BOX TESTING - B&C COVERAGE

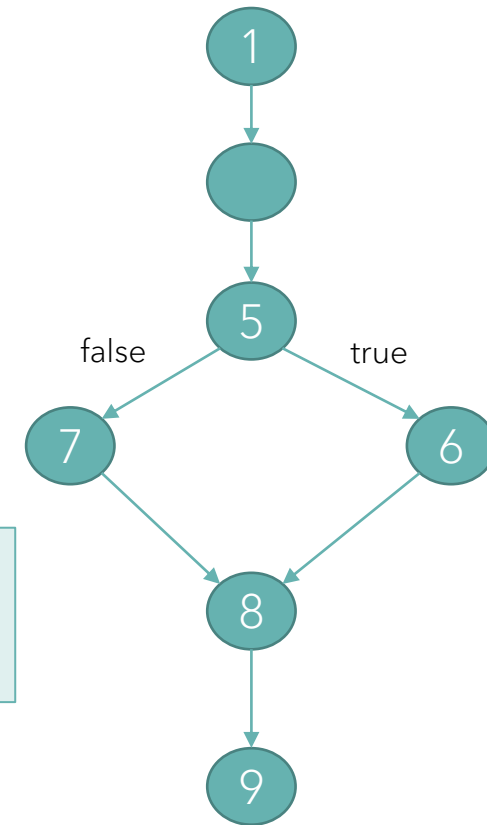
```
1 func() {  
2     int x, y;  
3     read(x);  
4     read(y)  
5     if( x == 0 || y > 0)  
6         y = y/x;  
7     else x = y + 2;  
8     write(x);  
9     write(y)  
10 }
```

Test case # 1
x = 0, y = -5

Test case # 2
x = 5, y = 5

Test case # 3
x = 3, y = -2

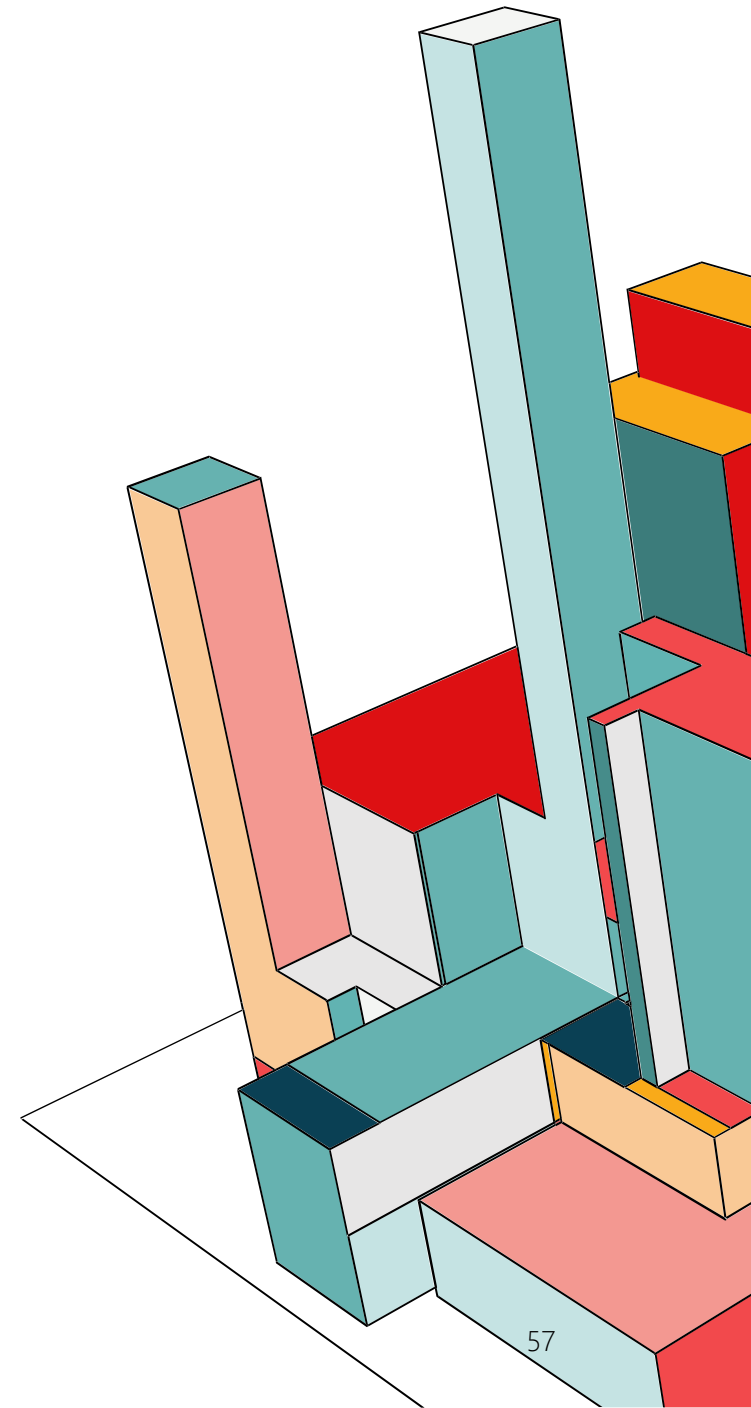
Branch&Condition coverage: 100%



WHITE-BOX TESTING

Types of coverage

- Statement coverage
- Branch coverage
- Condition coverage
- B&C coverage
- Path coverage
- Multiple condition coverage
- Finite state machine coverage
-



BLACK-BOX TESTING

- Based on specification
- Focus on input domain
 - Calculator: input domain is every number
 - Translator: input domain is every word or sentence



How to select the test input data?

BLACK-BOX TESTING - TEST DATA SELECTION

Exhaustive Testing

- Test every possible input
- Not feasible w.r.t. time

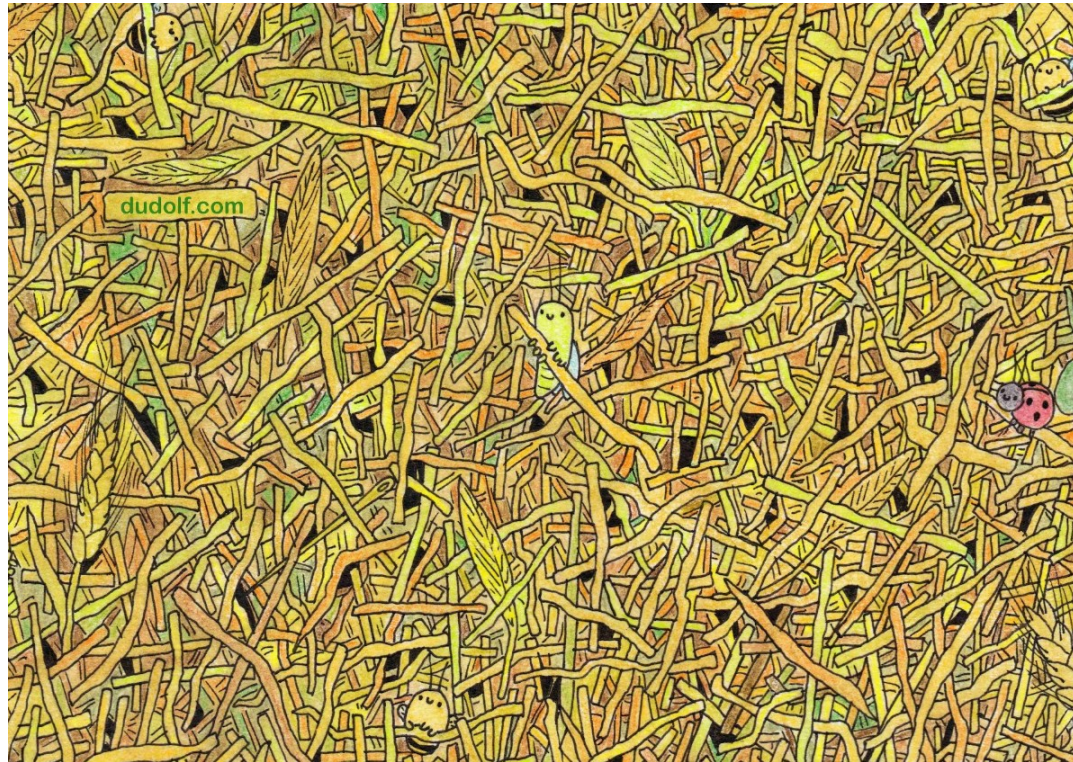
Example: `add(int a, int b)`

- $2^{32} * 2^{32} = 2^{64}$ or 10^{19} tests
- Assume 1 test / nanosecond
~ 10^9 tests / second
- 10^{10} seconds for exhaustive testing

317 years

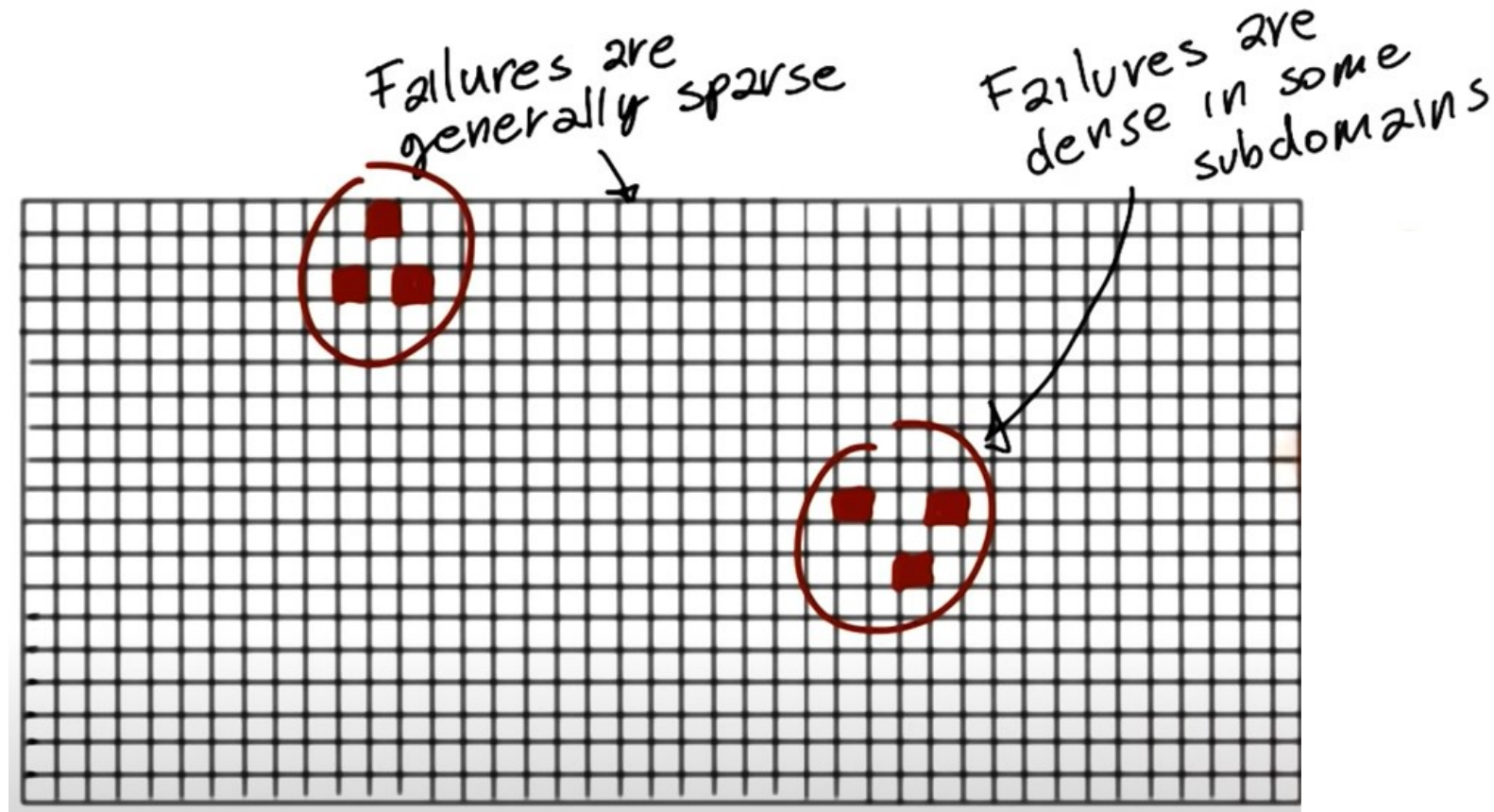
BLACK-BOX TESTING - TEST DATA SELECTION

Random Testing: Pick inputs randomly



Pick a test that reveals the bug is like finding a needle in a haystack

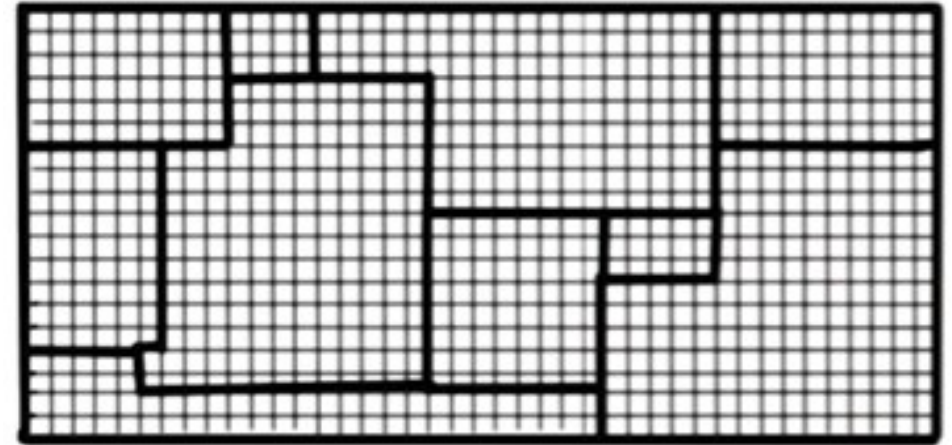
BLACK-BOX TESTING - TEST DATA SELECTION



BLACK-BOX TESTING - TEST DATA SELECTION

Partition Testing:

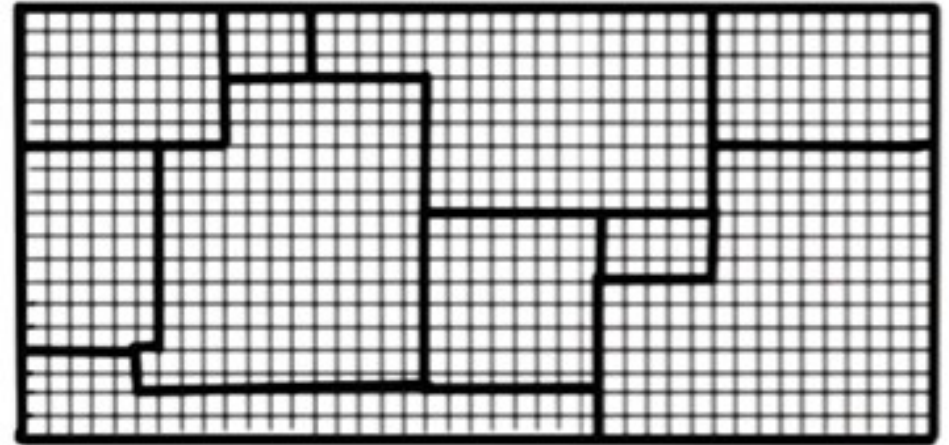
- This method divides the input data of software into different equivalence data classes (等价类).
- In general, input could be partitioned into valid/invalid equivalence class
- Test inputs can be selected from each partition to reduce time required for testing



BLACK-BOX TESTING - TEST DATA SELECTION

Equivalence Partition Hypothesis:

- If one condition/value in a partition passes all others will also pass.
- Likewise, if one condition in a partition fails, all other conditions in that partition will fail.



EXAMPLE

Enter OTP

*Must include six digits

Equivalence Partitioning			
Invalid	Invalid	Valid	Valid
Digits \geq 7	Digits \leq 5	Digits=6	Digits=6
67545678	9754	654757	213309

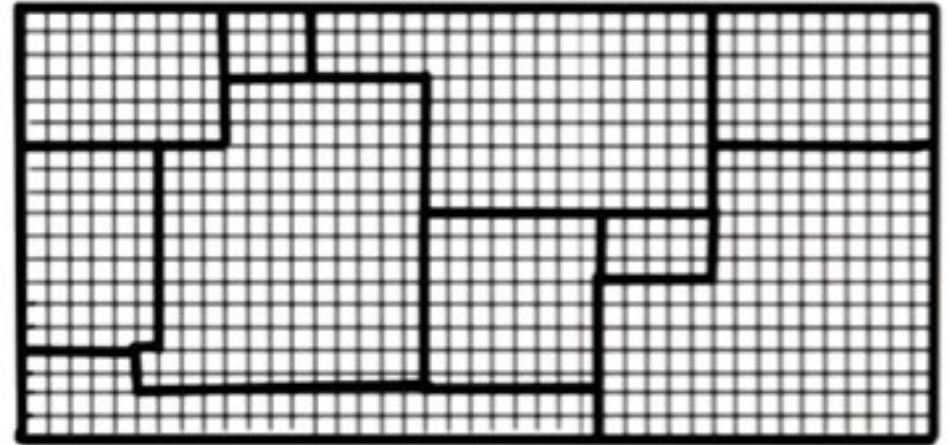
Still a lot of possibilities for the valid partition!

<https://www.geeksforgeeks.org/equivalence-partitioning-method/>

BLACK-BOX TESTING - TEST DATA SELECTION

Boundary Values:

- Errors tend to occur at boundary of a (sub)domain
- Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values, which increases the chance of revealing faults



EXAMPLE

Order Pizza: 1

Submit

Specification

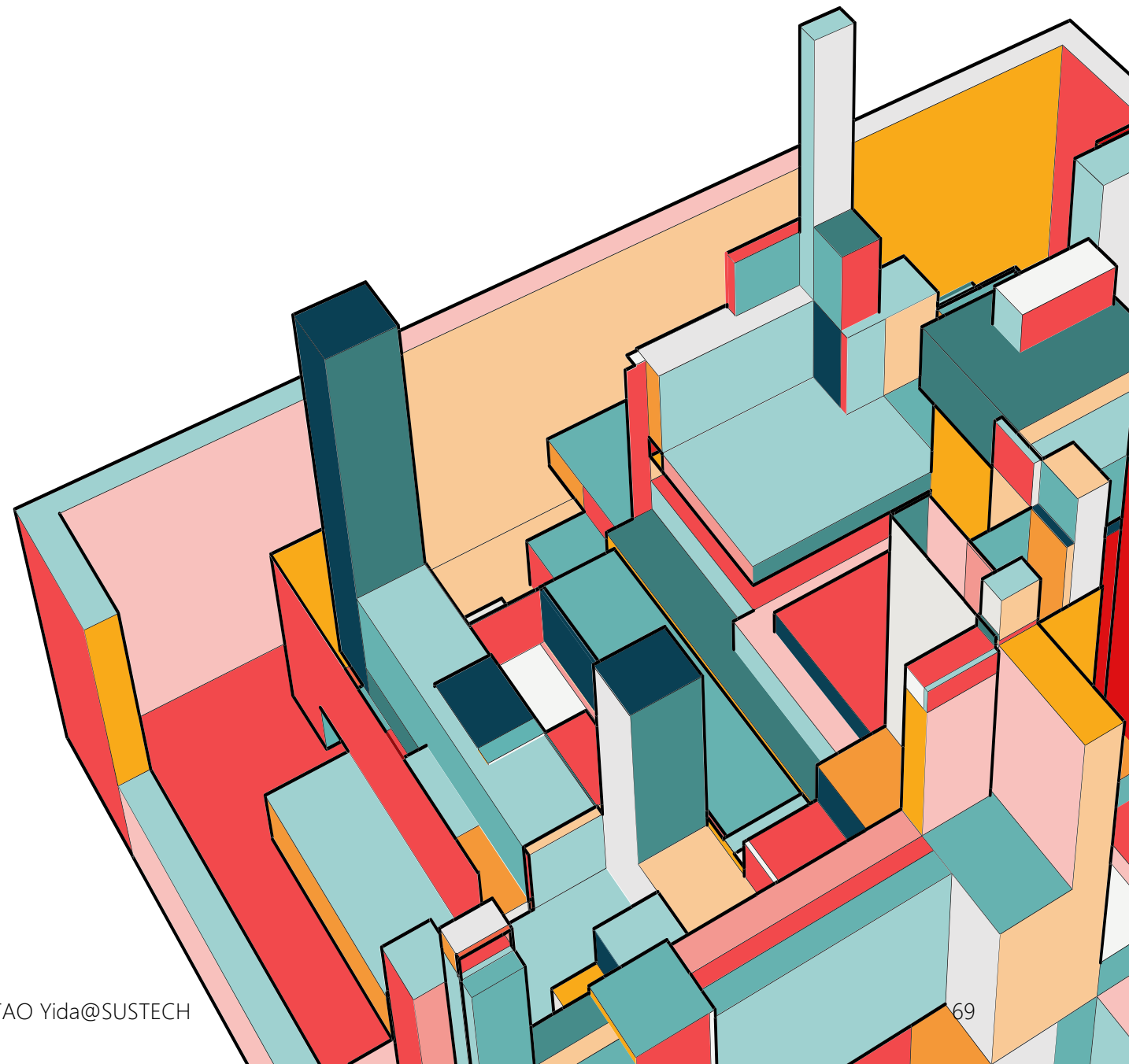
- Pizza values 1 to 10 is considered valid. A success message is shown.
- Pizza value 11 to 99 are considered invalid for order and an error message will appear, “Only 10 Pizza can be ordered”



<https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html>

READINGS

- Chapter 11-13. Software Engineering at Google by Winters et al
- 第9章 软件测试. 现代软件工程基础 by 彭鑫 et al.



NEXT

- Software Testing II