



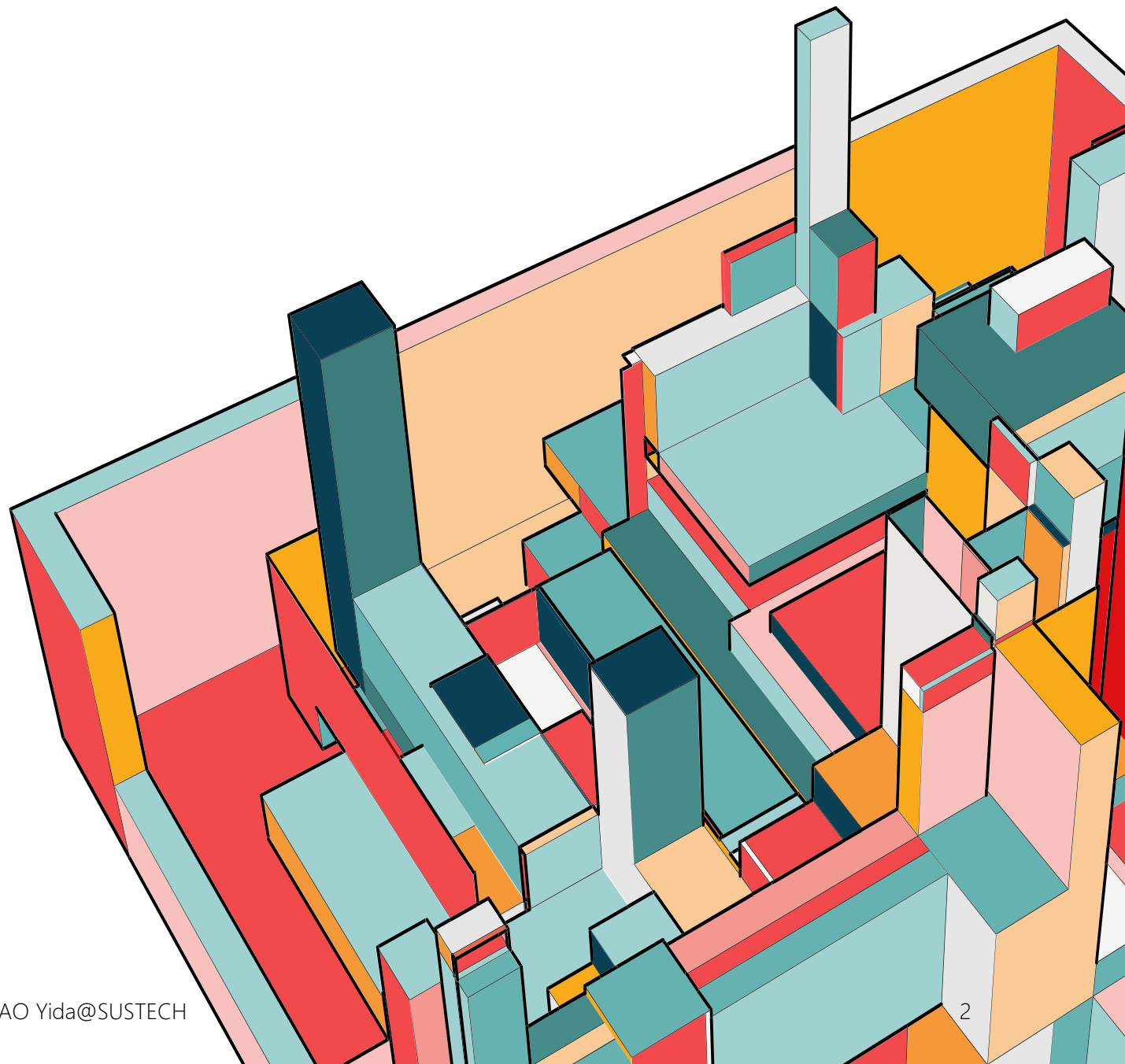
CS304 SOFTWARE ENGINEERING

Yida Tao

taoyd@sustech.edu.cn

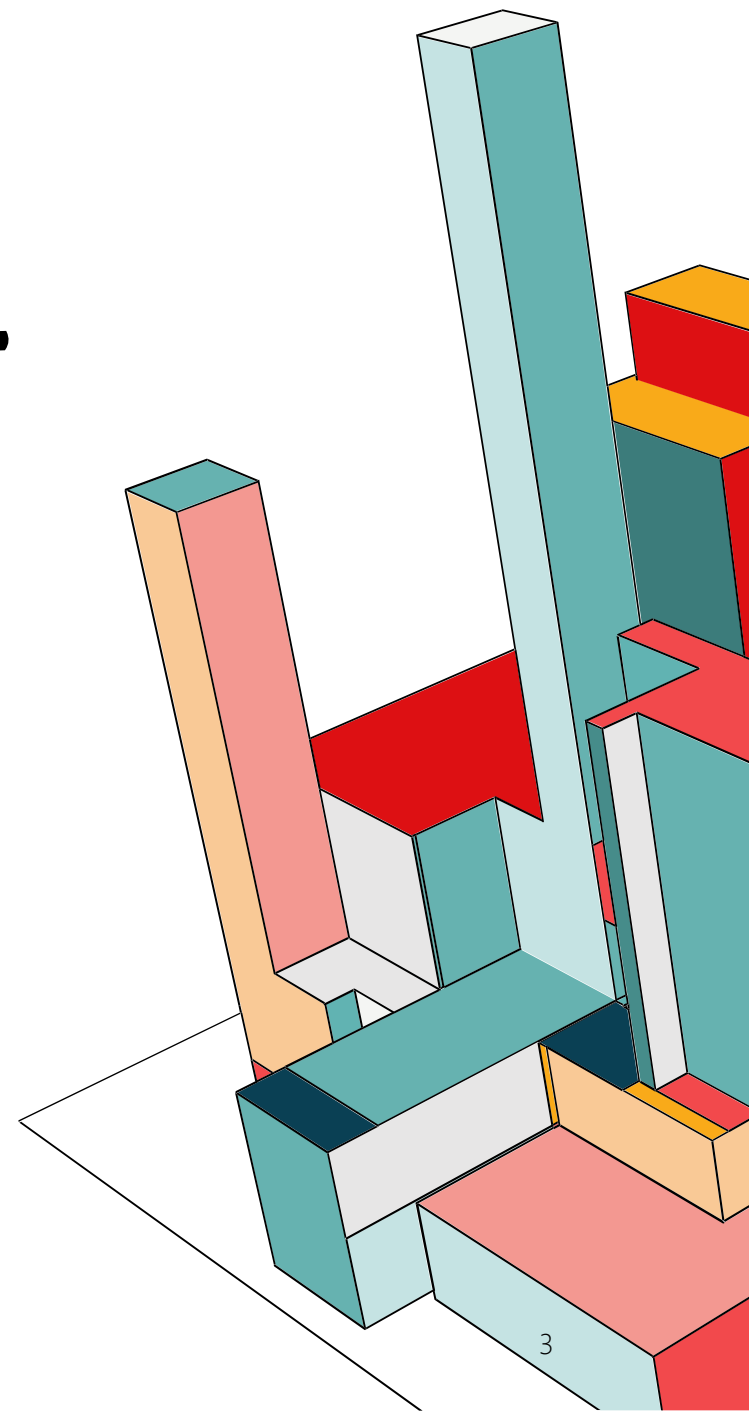
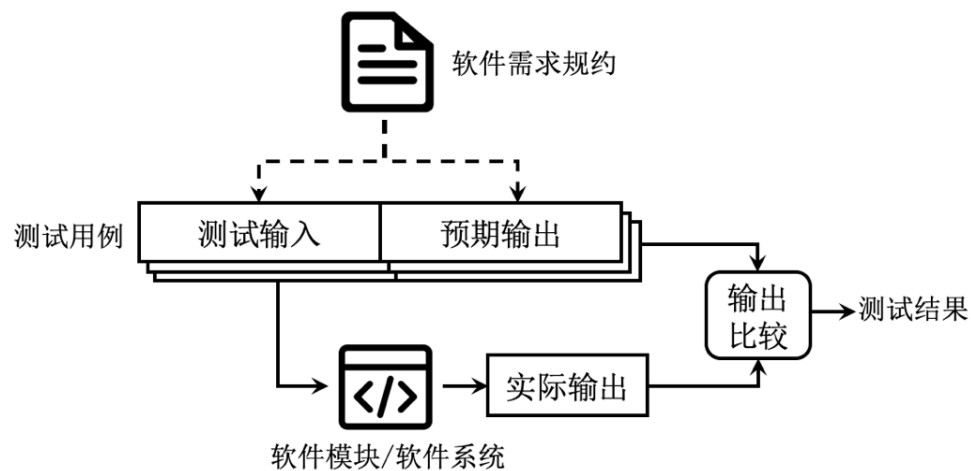
LECTURE 11

- Overview
- Testing Concepts
- Blackbox & Whitebox Testing
- Test Doubles
- Writing Maintainable Tests



SOFTWARE TESTING IN A NUTSHELL

- We got a software
- We throw some sample data 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

Manual testing won't scale for modern software development

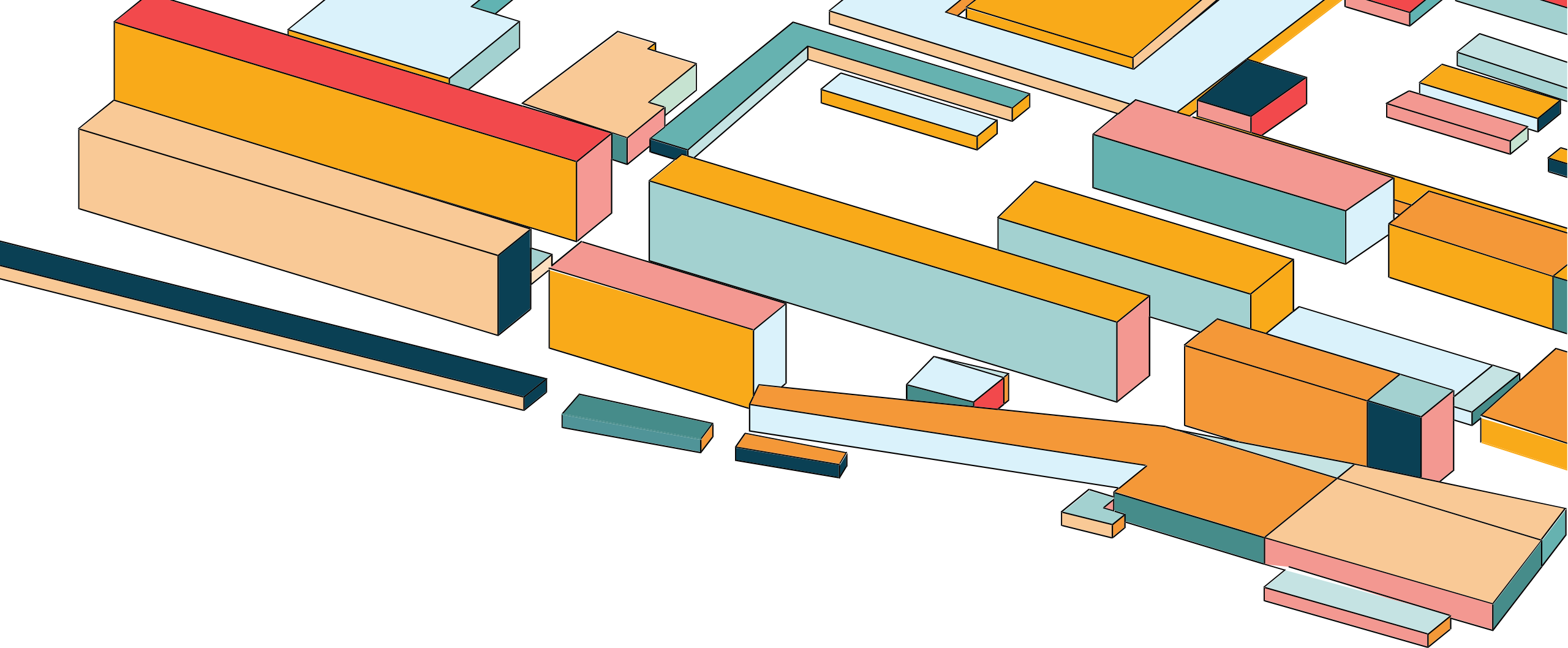
- Thousands of lines of code
- Hundreds of libraries and frameworks
- Multiple platforms & languages
- Uncountable number of configurations
- Frequent updates or new versions
- Delivered via unreliable networks

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.



TESTING CONCEPTS

TEST CASE VS. TEST SUITE

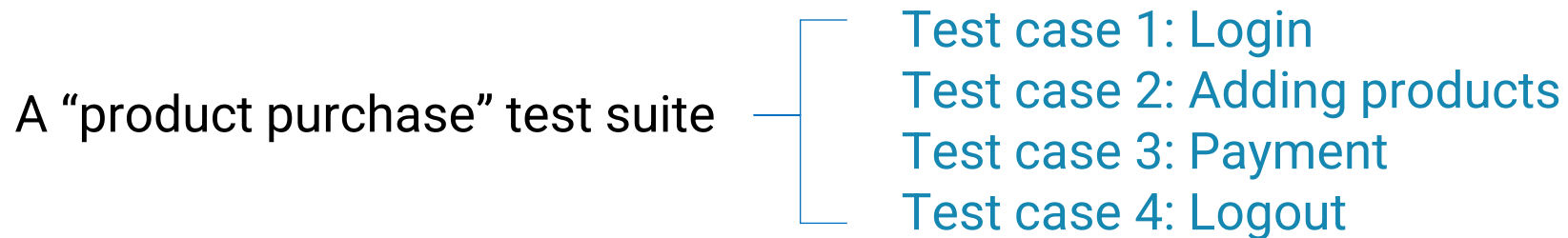
- **Test case** (测试用例): a sequence of actions necessary to verify a specific functionality or feature of the software.
- A test case specifies the prerequisites, post conditions, steps, and data required for feature verification.

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

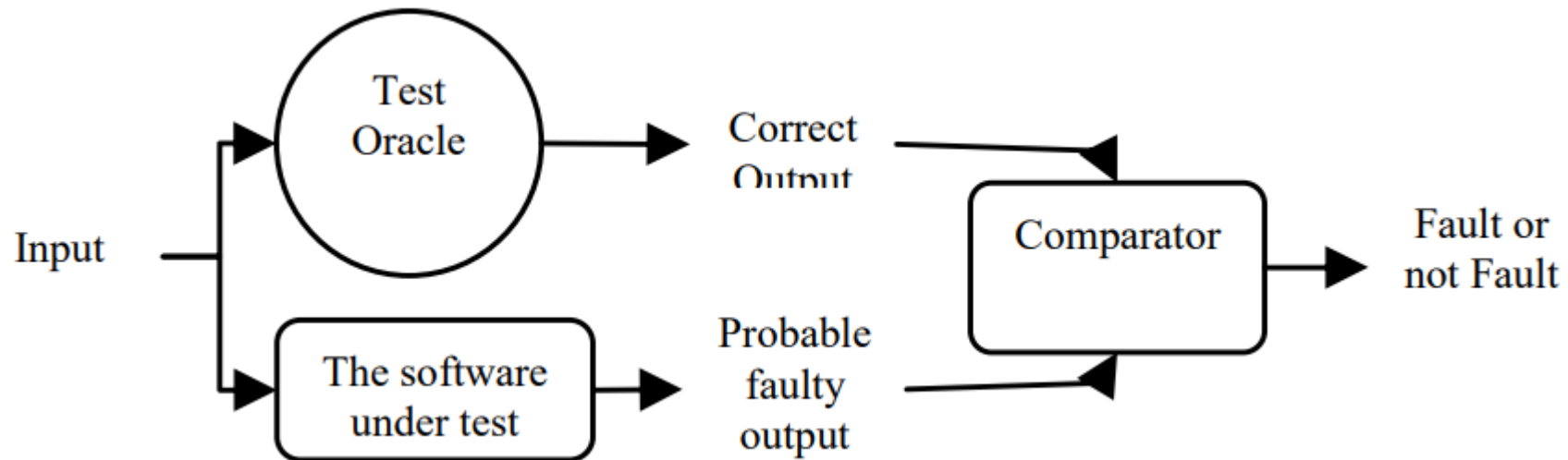
- **Test suites** (测试套件、测试集) are the logical grouping or collection of test cases to run a single job with different test scenarios.
- Example: a test suite for product purchase has multiple test cases



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

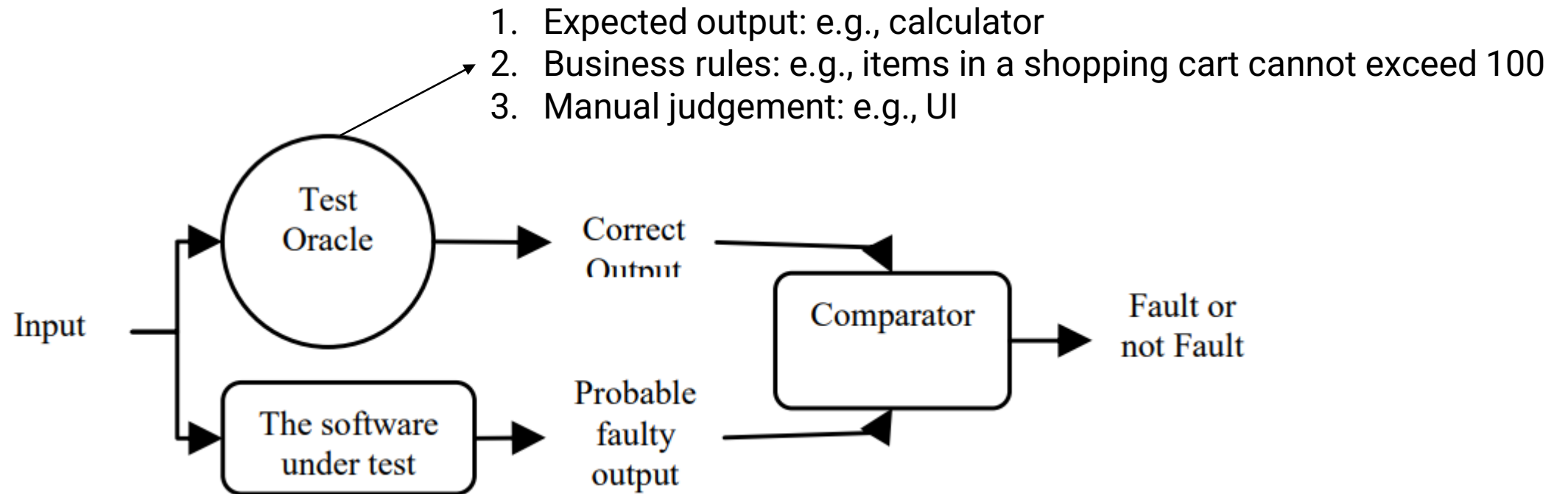
TEST INPUT VS. TEST ORACLE

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



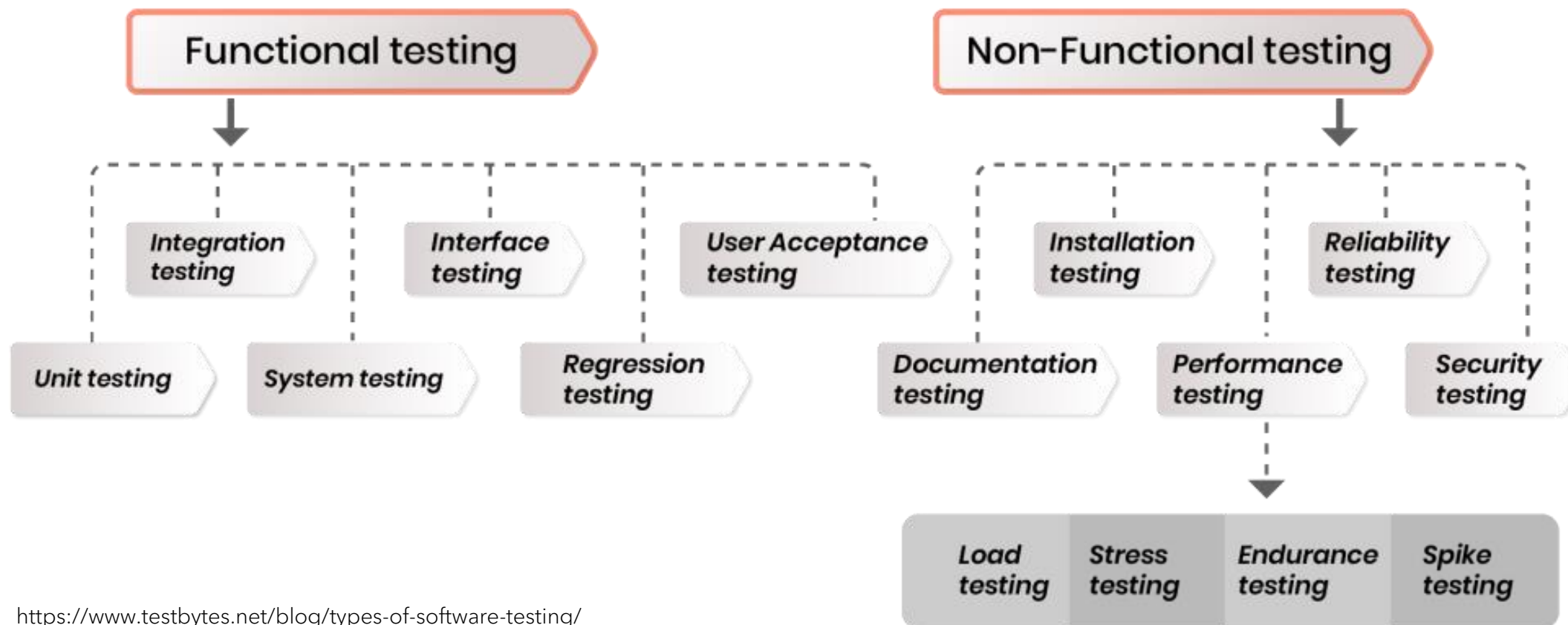
Kadir, W.M., & Nasir, W.M. (2008). Intelligent and automated software testing methods classification.

TEST INPUT VS. TEST ORACLE



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

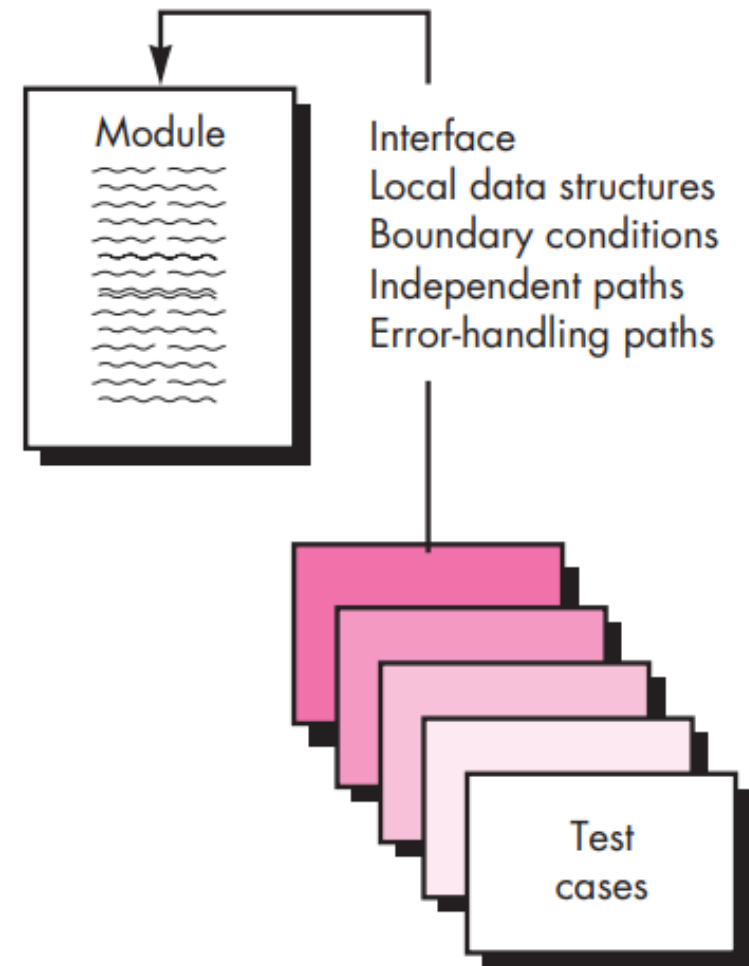
TYPES OF SOFTWARE TESTING



<https://www.testbytes.net/blog/types-of-software-testing/>

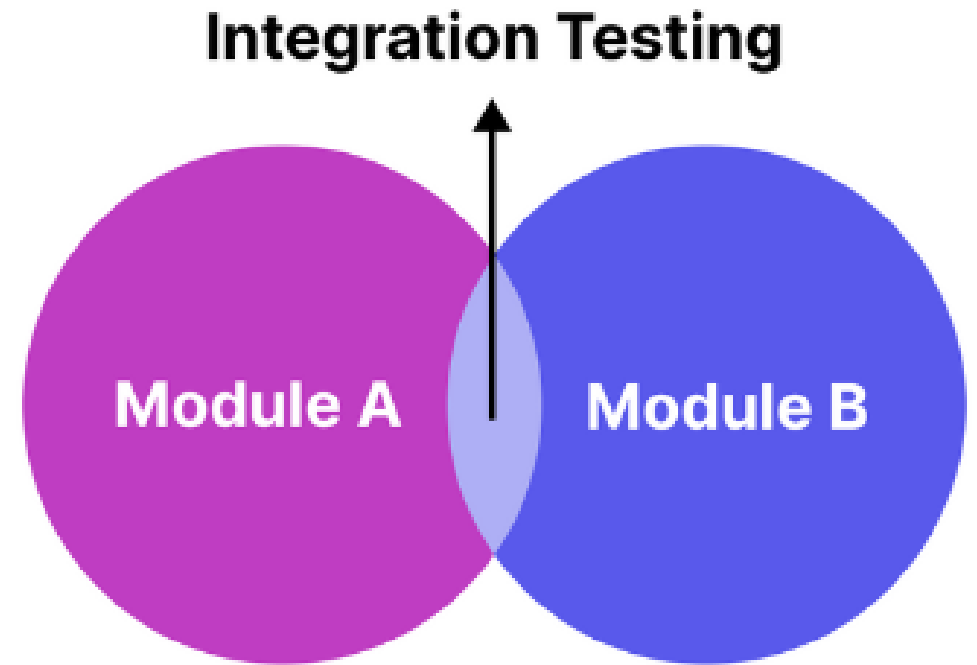
UNIT TESTING

- 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

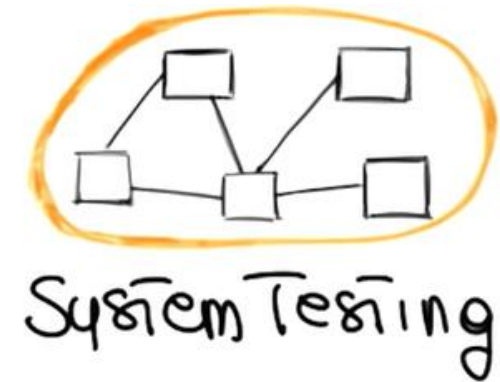
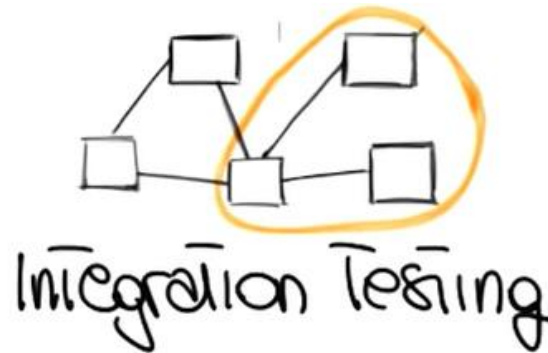
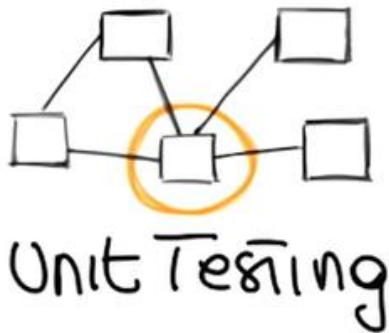
Test Case ID	Test Case Objective	Test Case Description	Expected Result
1	Check the interface link between the Login and Mailbox module	Enter login credentials and click on the Login button	To be directed to the Mail Box
2	Check the interface link between the Mailbox and Delete Mails Module	From Mailbox 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

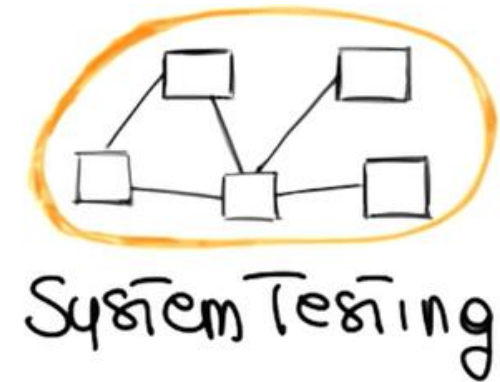
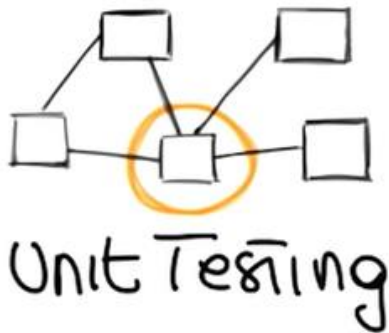
- System testing aims at testing that the product fulfills the business specifications and is ready for deployment
- System testing is generally executed after the integration test and before the delivery of the software product
- System testing is often conducted in an environment closely resemble the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



<https://youtu.be/SjVfvXJeajA>

SYSTEM TESTING

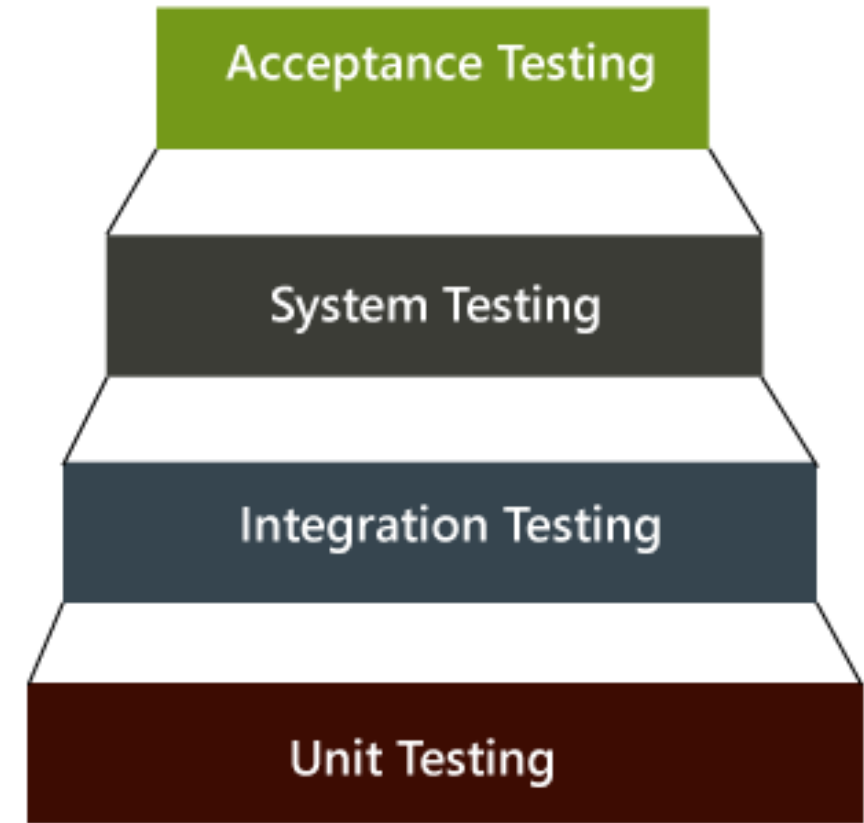
- System testing is usually performed by a dedicated testing team
- System testing may include functional testing, performance testing, security testing, etc.



<https://youtu.be/SjVfvXJeajA>

ACCEPTANCE TESTING

- The last phase of software testing performed after System Testing and before making the system available for actual use.
- Acceptance testing is used to determine whether the product is working for the end-users correctly
- Acceptance testing may involve end users perform realistic tasks using real data

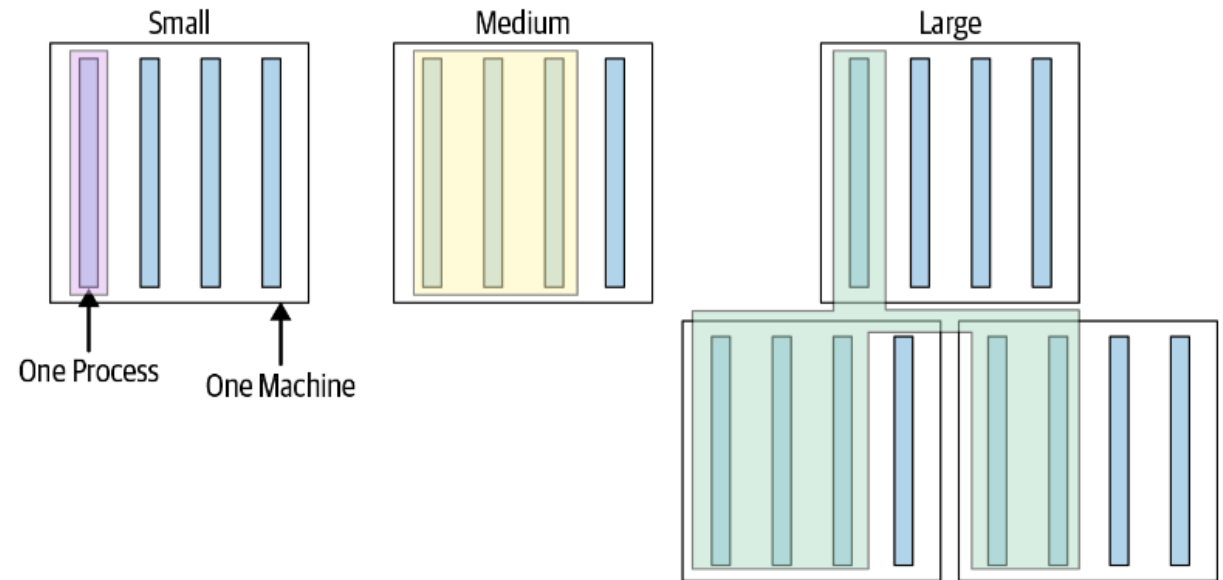


TEST SIZE

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

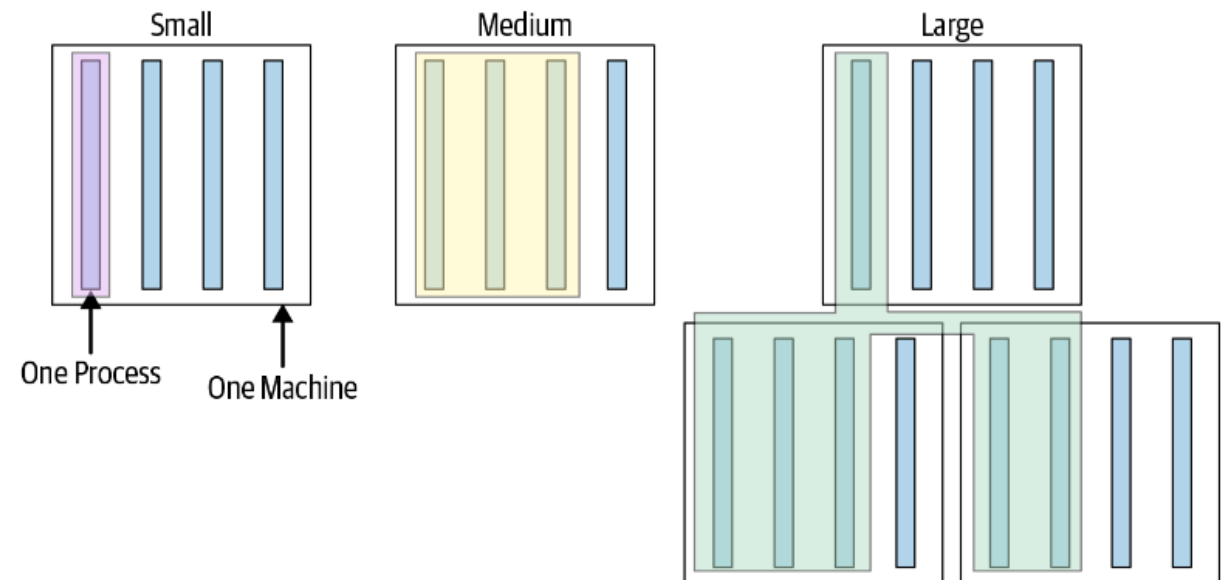


TEST SIZE

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.)

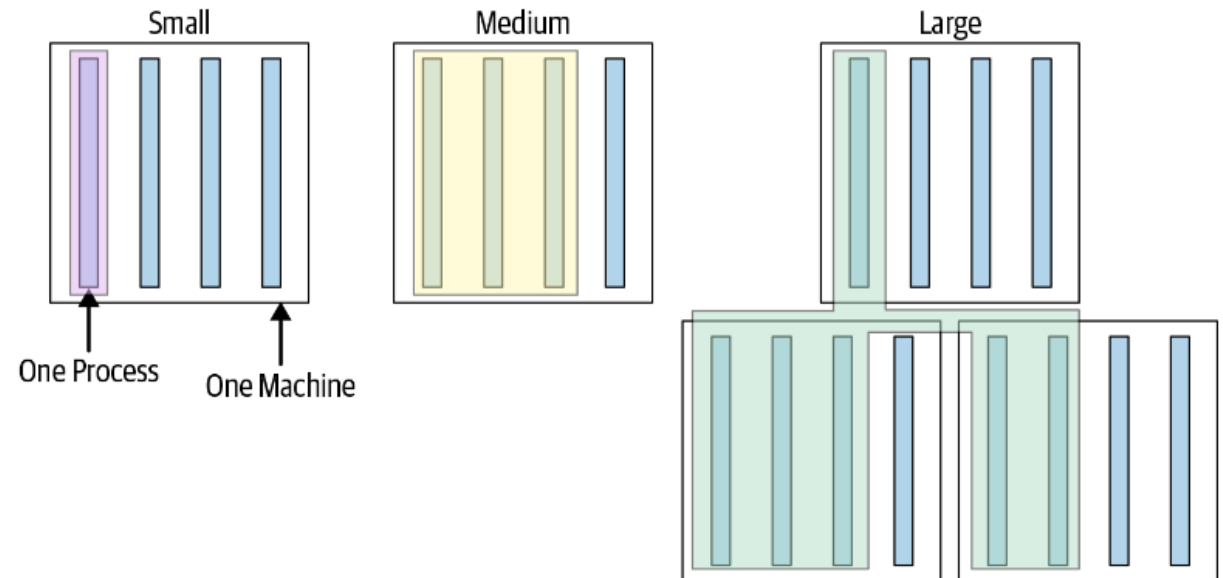


TEST SIZE

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



TEST SCOPE

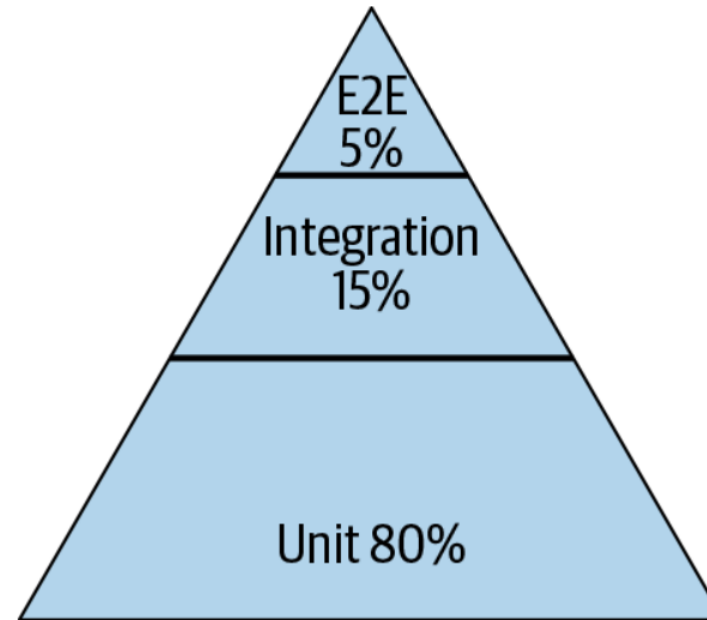
How much code a test intends to validate

- Narrow-scoped tests
 - E.g., unit tests
 - Validate the logic in a small, focused part of the codebase (e.g., class, method)
- Medium-scoped tests
 - E.g., integration tests
 - Verify interactions between a small number of components (e.g., server + database)
- Large-scoped tests
 - E.g., end-to-end tests or system tests
 - Validate several parts of the system or the entire system

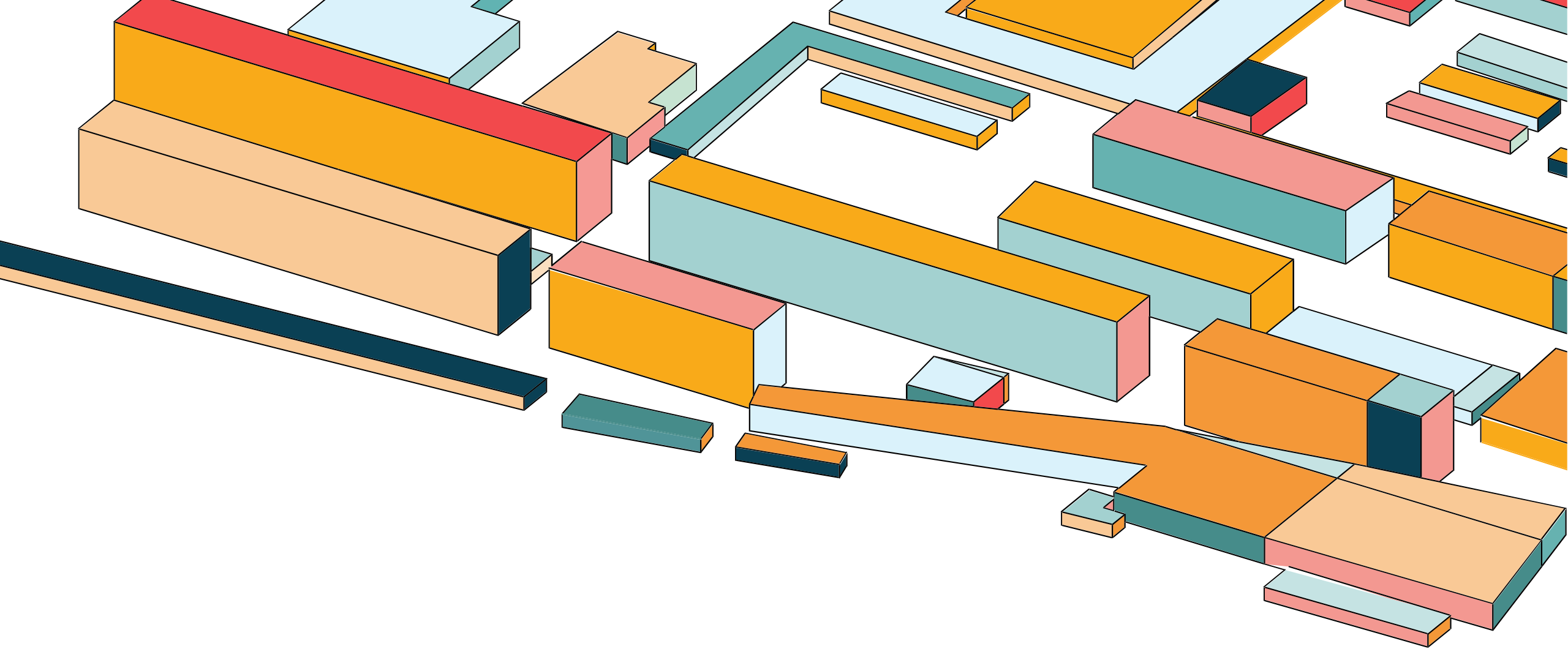
TEST SCOPE

How much code a test intends to validate

- 80% of tests being narrow-scoped unit tests that validate the majority of business logic
- 15% medium-scoped integration tests that validate the interactions between two or more components
- 5% end-to-end tests that validate the entire system.



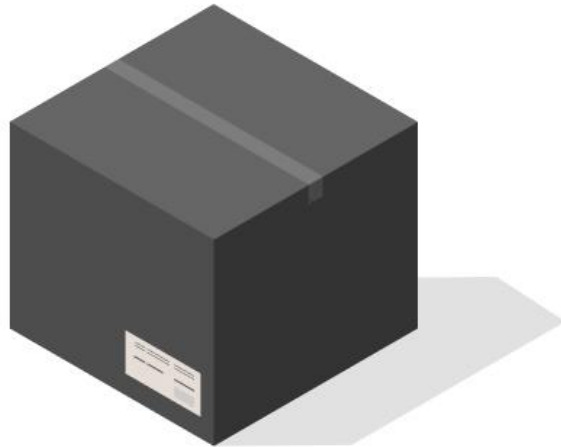
Google's test pyramid



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

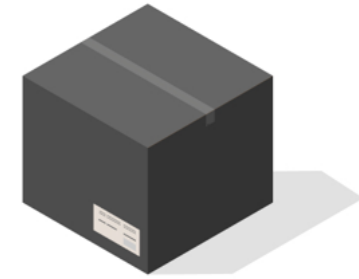


White-box testing

- 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

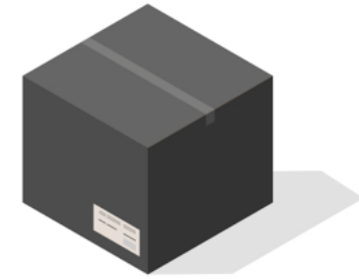
Miss the bug

EXAMPLE II

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

Possible implementation

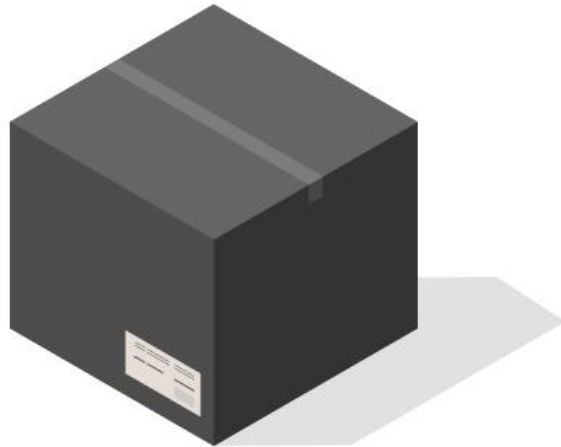
```
int func(int num){  
    int result;  
    result = num/2;  
    return result;  
}
```



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 }
```

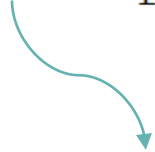
Test case # 1
a = 1, b = 2

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)

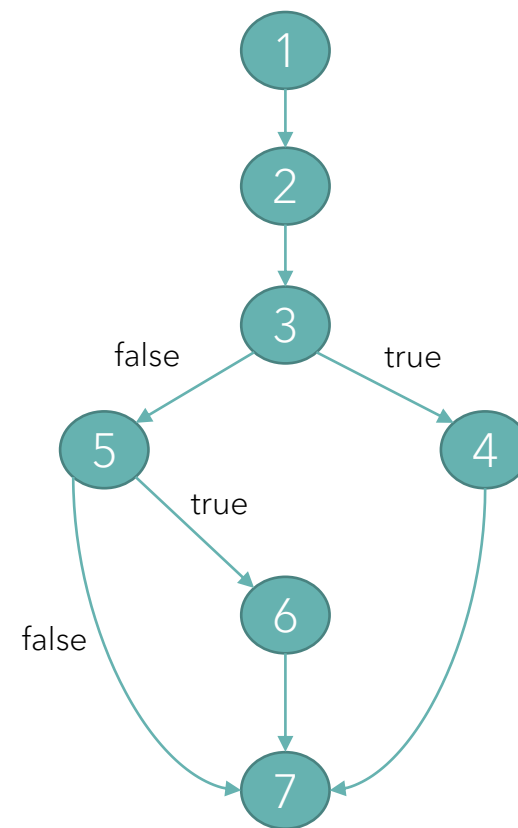
Branch coverage = (# of executed branches / # of total branches) x 100%

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:
 $1 / 4 = 25\%$

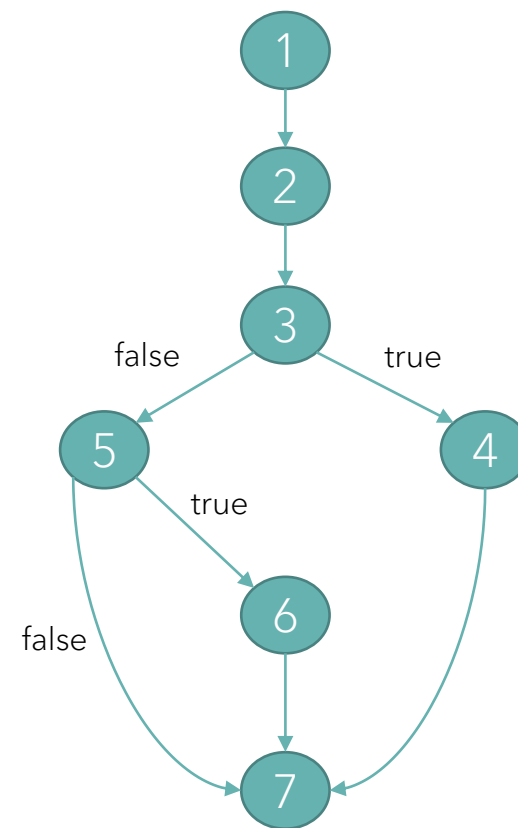


WHITE-BOX TESTING - BRANCH COVERAGE

```
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:
 $2 / 4 = 50\%$



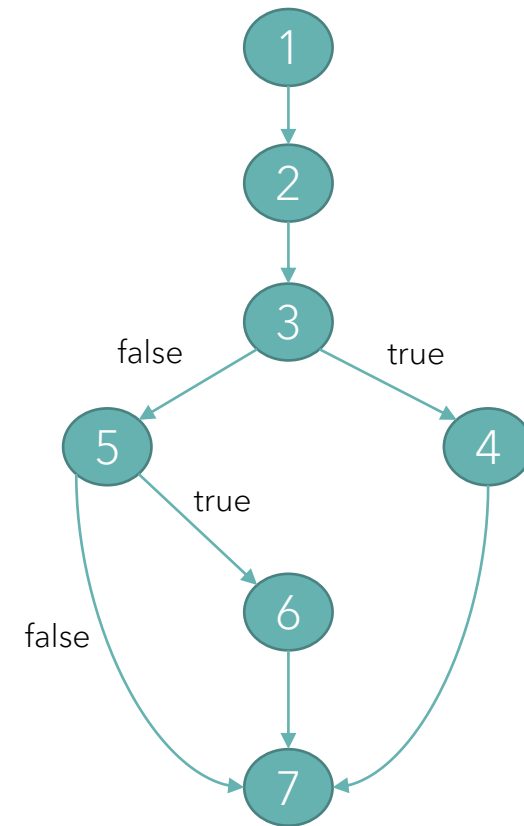
WHITE-BOX TESTING - BRANCH COVERAGE

```
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

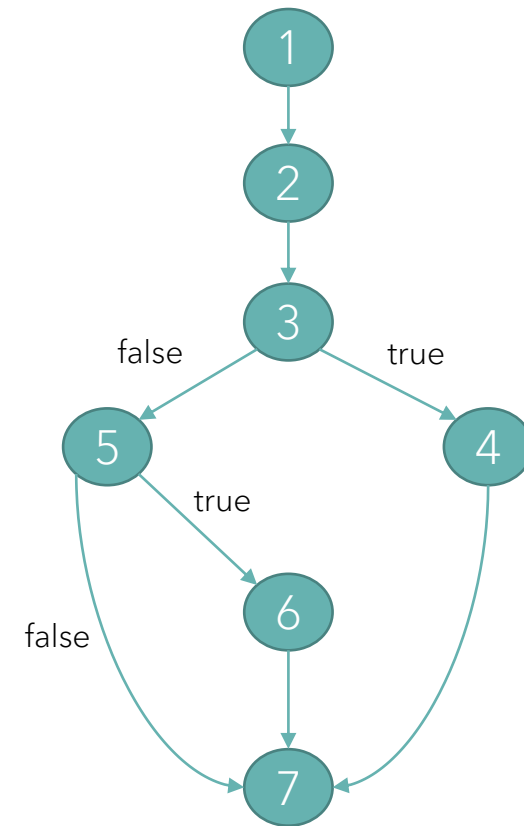
```
1 sum(int a, int b){  
2     int result = a+b;  
3     if(result>0)  
4         print("positive" + result);  
5     else if(result<0)  
6         print("negative" + result);  
7 }
```

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

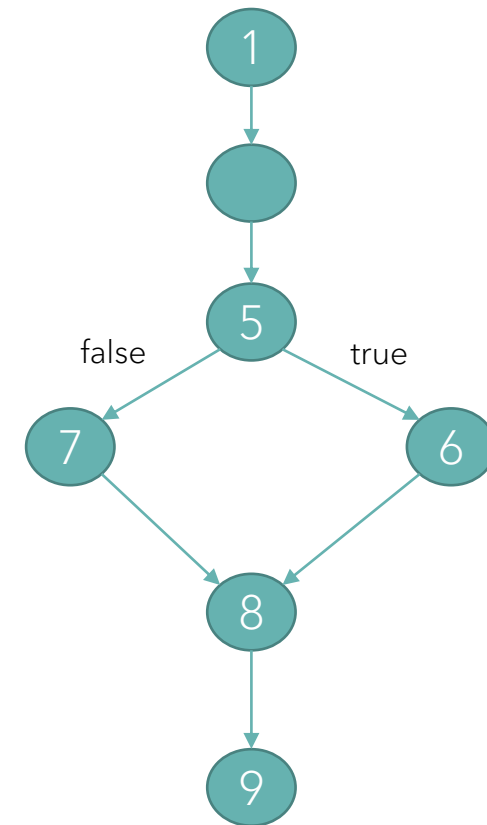
```
1 func() {  
2     int x, y;  
3     read(x);  
4     read(y)  
5     if( x == 0 || y > 0)  
6         y = y/x;  
7     else x = y + 2;  
8     write(x);  
9     write(y)  
10 }
```

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

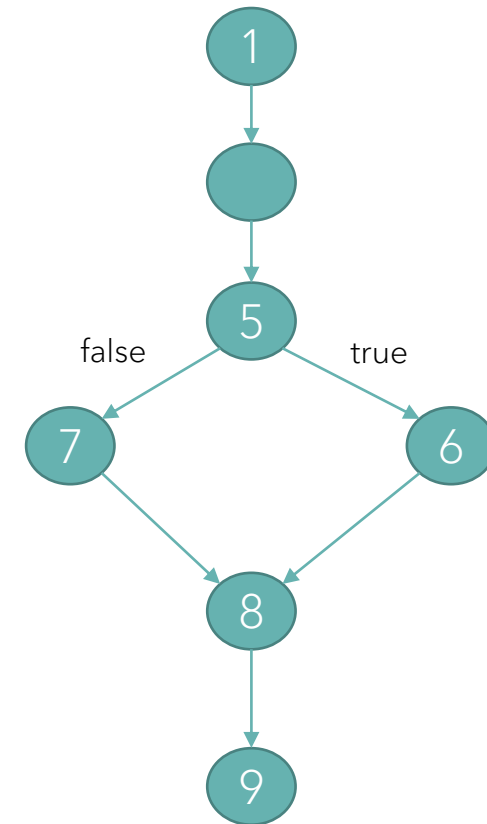
```
1 func() {  
2     int x, y;  
3     read(x);  
4     read(y)  
5     if( x == 0 || y > 0)  
6         y = y/x;  
7     else x = y + 2;  
8     write(x);  
9     write(y)  
10 }
```

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

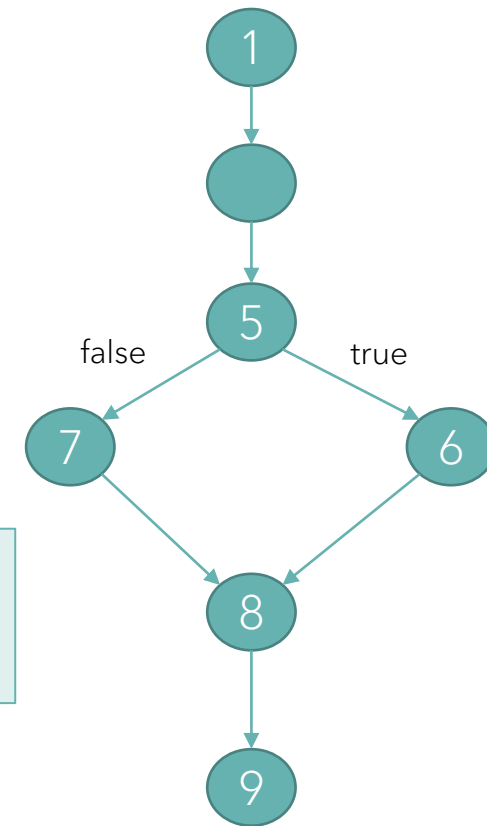
```
1 func() {  
2     int x, y;  
3     read(x);  
4     read(y)  
5     if( x == 0 || y > 0)  
6         y = y/x;  
7     else x = y + 2;  
8     write(x);  
9     write(y)  
10 }
```

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

Test case # 2
x = 5, y = 5

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

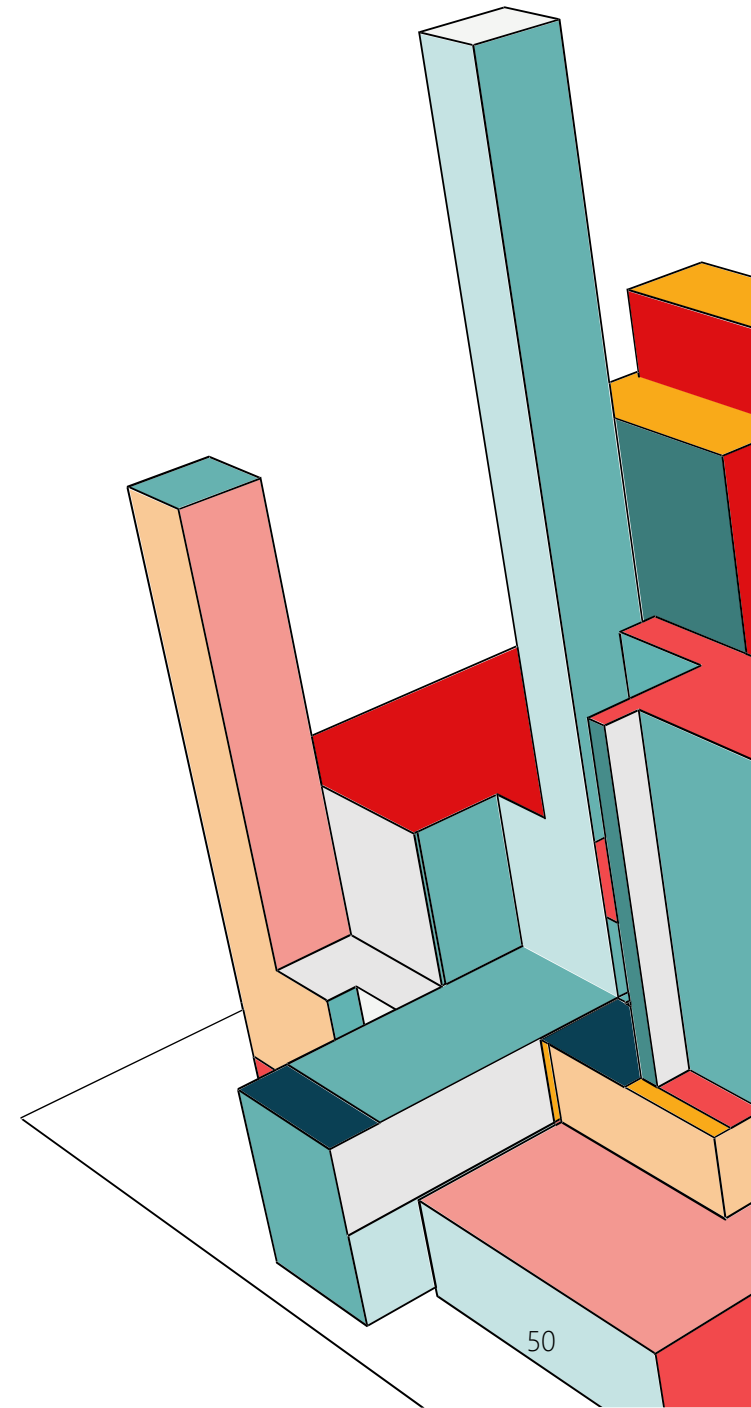
Branch&Condition coverage: 100%



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

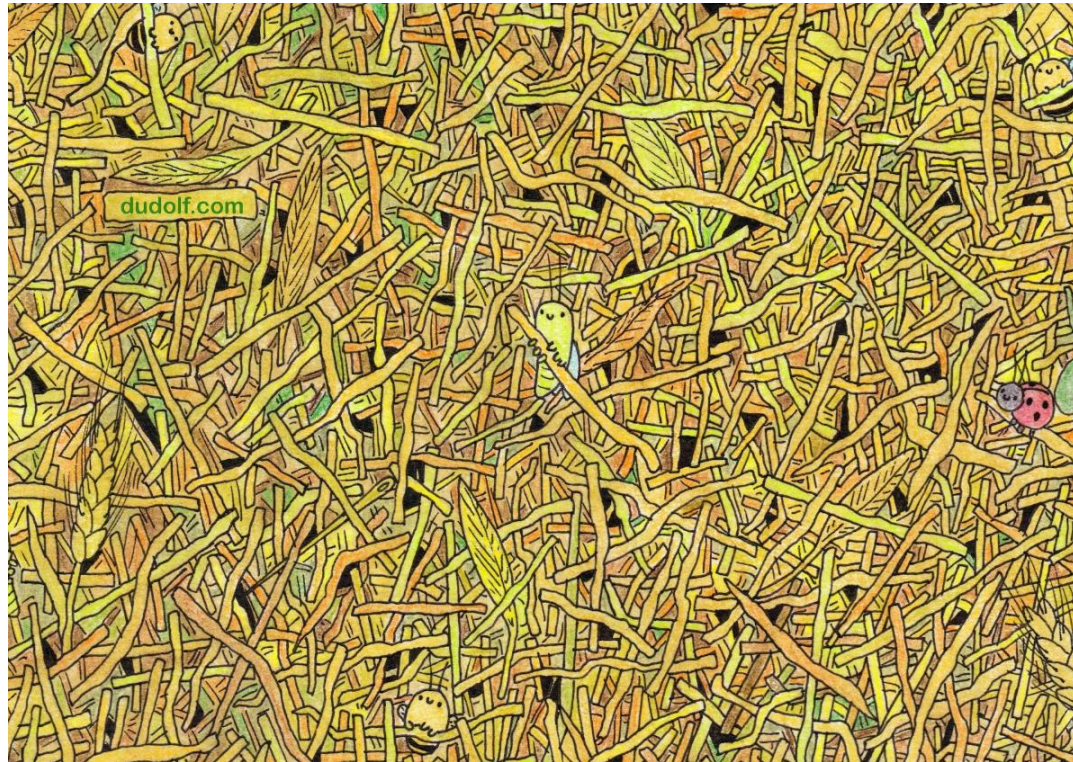
Example: `add(int a, int b)`

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

317 years

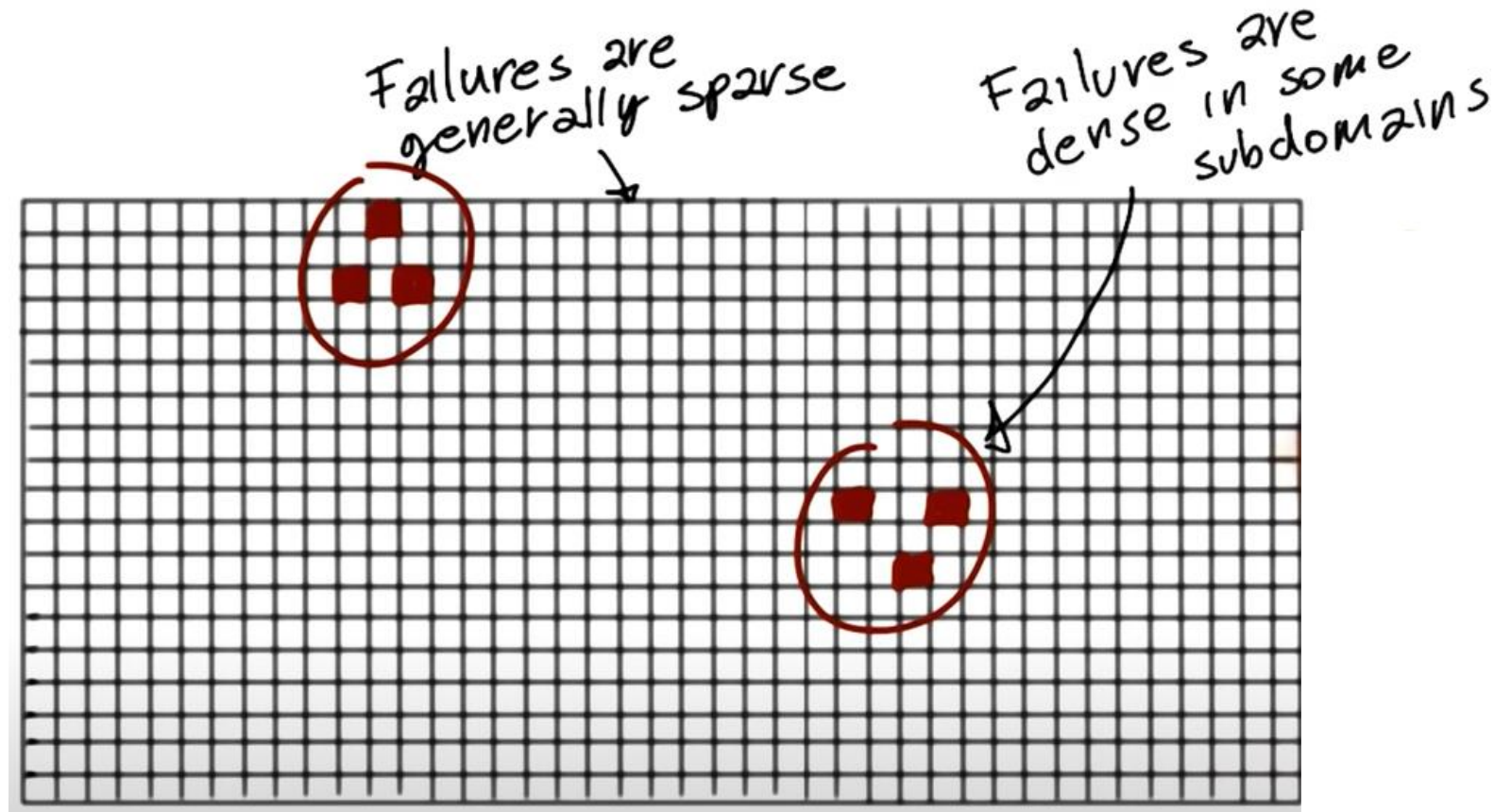
BLACK-BOX TESTING - TEST DATA SELECTION

Random Testing: Pick inputs randomly



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

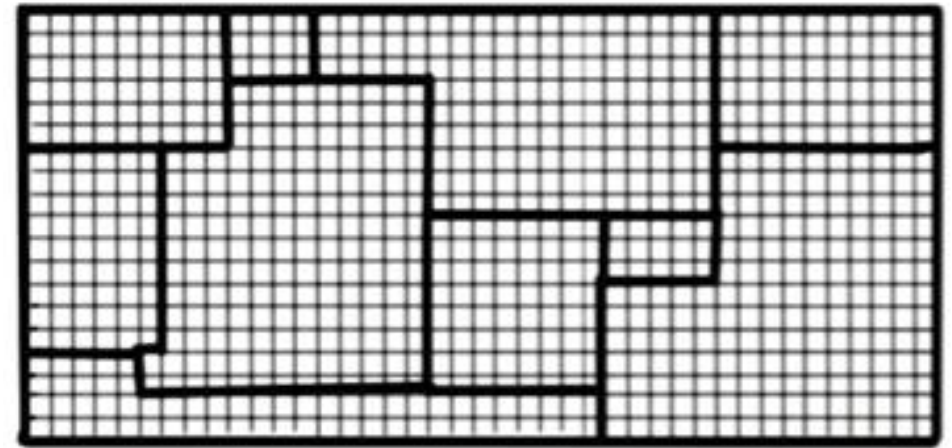
BLACK-BOX TESTING - TEST DATA SELECTION



BLACK-BOX TESTING - TEST DATA SELECTION

Partition Testing:

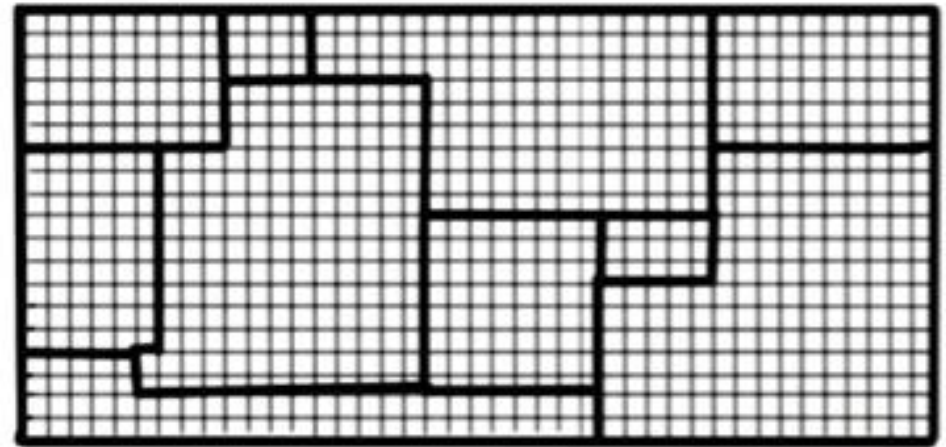
- This method divides the input data of software into different equivalence data classes (等价类).
- In general, input could be partitioned into valid/invalid equivalence class
- Test inputs can be selected from each partition to reduce time required for testing



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.



基于等价类的货品登记程序测试用例设计

输入货品信息，最后给出货品存放指示

- ✓ 编号必须为英文字母与数字的组合，由字母开头，包含6个字符，且不能有特殊字符
- ✓ 货品的登记数量在10到500之间（包含10和500）
- ✓ 货品的类型是设备、零件、耗材中的一种
- ✓ 货品的尺寸是大型、中型、小型中的一种
- ✓ 违反以上要求的登记信息被视为无效输入

大型货品存放在室外堆场，中型货品存放在专用仓库，小型货品存放在室内货架

输入数据	有效等价类	无效等价类
货品编号	(1) 符合规则的编号	(7) 编号长度不为6 (8) 编号有特殊字符 (9) 编号不以字母开头
登记数量	(2) $10 \leq \text{数量} \leq 500$	(10) 数量 < 10 (11) 数量 > 500
货品类型	(3) {设备, 零件, 耗材}	(12) 非设备、零件、耗材中的一种
货品尺寸	(4) 大型 (5) 中型 (6) 小型	(13) 非大型、中型、小型中的一种

基于等价类的货品登记程序测试用例设计

输入数据	预期输出	覆盖的等价类
A=EQ0101, B=30, C=设备, D=大型	合法登记信息, 存放地为室外堆场	(1) (2) (3) (4)
A=CM0202, B=100, C=零件, D=中型	合法登记信息, 存放地为专用仓库	(1) (2) (3) (5)
A=MT0303, B=400, C=耗材, D=小型	合法登记信息, 存放地为室内货架	(1) (2) (3) (6)
A=EQ01023, B=30, C=设备, D=大型	非法登记信息	(7)
A=EQ0102#, B=30, C=设备, D=大型	非法登记信息	(8)
A=0102EQ, B=30, C=设备, D=大型	非法登记信息	(9)
A=EQ0101, B=0, C=设备, D=大型	非法登记信息	(10)
A=EQ0101, B=600, C=设备, D=大型	非法登记信息	(11)
A=EQ0101, B=30, C=装置, D=大型	非法登记信息	(12)
A=MT0202, B=100, C=耗材, D=中小型	非法登记信息	(13)

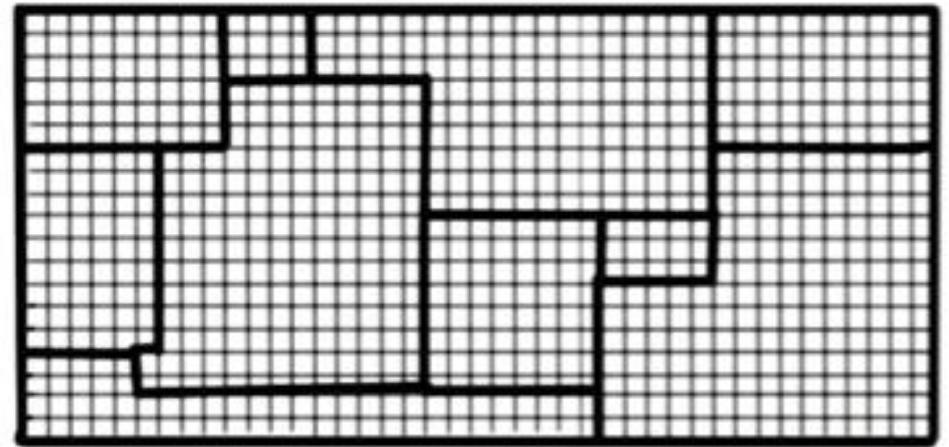
输入数据	有效等价类	无效等价类
货品编号	(1) 符合规则的编号	(7) 编号长度不为6 (8) 编号有特殊字符 (9) 编号不以字母开头
登记数量	(2) $10 \leq \text{数量} \leq 500$	(10) 数量 < 10 (11) 数量 > 500
货品类型	(3) {设备, 零件, 耗材}	(12) 非设备、零件、耗材中的一种
货品尺寸	(4) 大型 (5) 中型 (6) 小型	(13) 非大型、中型、小型中的一种

A: 货品编号, B: 登记数量, C: 货品类型, D: 货品尺寸

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



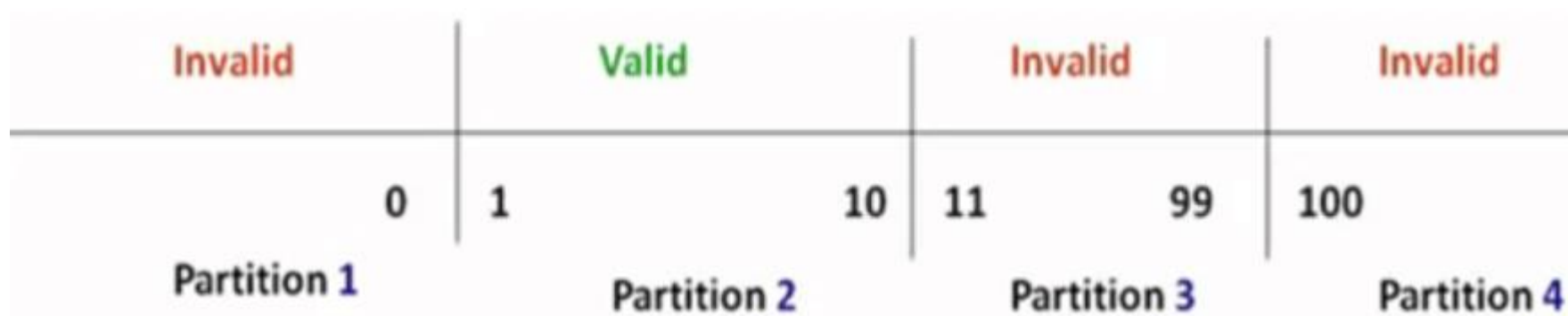
EXAMPLE

Order Pizza: 1

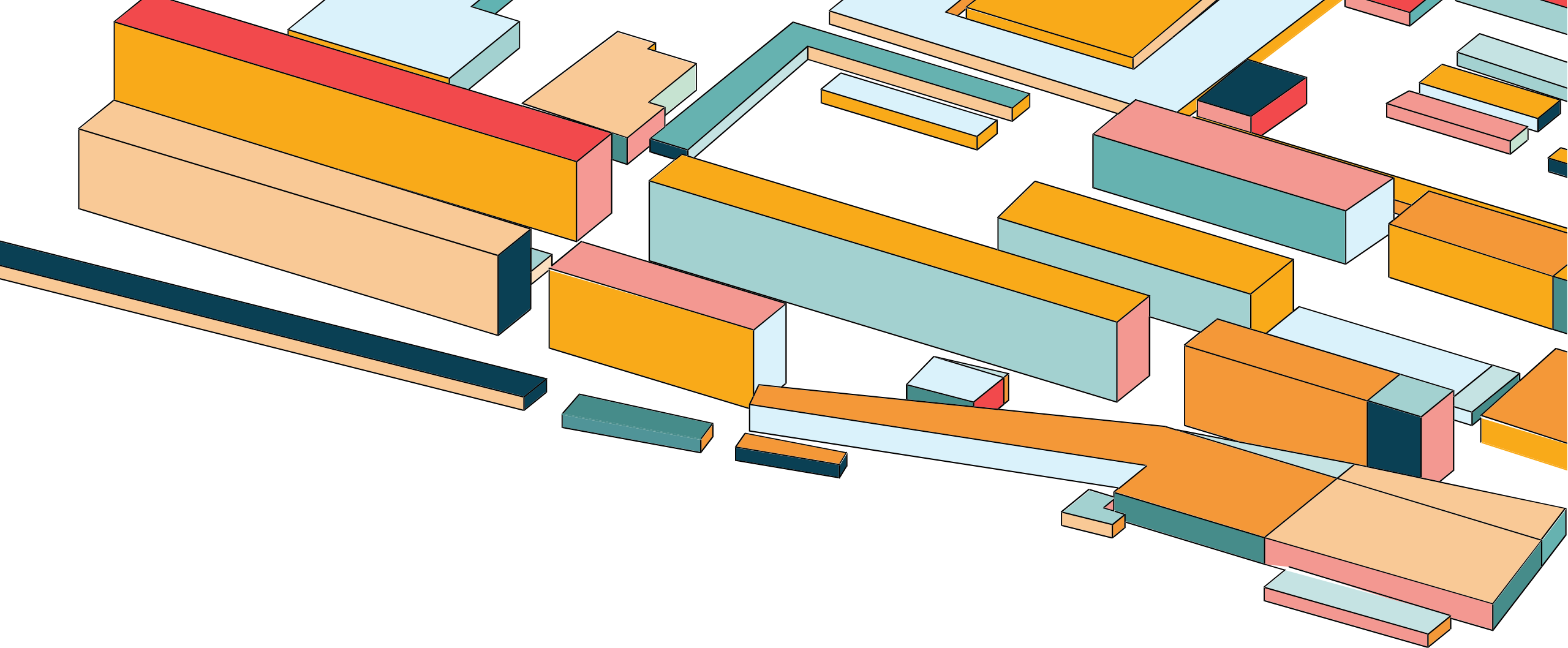
Submit

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>



TEST DOUBLES

WHY TEST DOUBLES?

- Imagine writing a test suite for a function that sends a request to an external server and then stores the response in a database.
- The test suite may
 - take hours to run
 - become flaky (不稳定) due to issues like random network failures or tests overwriting one another's data.
- Test doubles come in handy in such cases



A test double is an object or function that can stand in for a real implementation in a test, similar to how a stunt double can stand in for an actor in a movie.

TEST DOUBLES

- “Test doubles” are objects or code components that mimic the behavior of real objects or components in the system under test (SUT).
 - Test doubles are commonly used in unit testing to isolate the SUT and its dependencies, such as external services or libraries, by replacing them with lightweight, controllable substitutes.
- Fakes
 - Stubs
 - Mocks

FAKES

- A fake is an implementation that behaves "naturally", but is not "real"
- Fakes have a **pre-written implementation** of the object that they are supposed to represent
- Purpose: simplify the implementation of tests by **removing unnecessary or heavy dependencies**, usually used for performance reasons
- Examples
 - In-memory database: You would never use this for production (since the data is not persisted), but it's perfectly adequate as a database to use in a testing environment.

<https://stackoverflow.com/a/55030455/636398>

FAKES

- A fake is an implementation that behaves "naturally", but is not "real"
- Fakes have a **pre-written implementation** of the object that they are supposed to represent
- Purpose: simplify the implementation of tests by **removing unnecessary or heavy dependencies**, usually used for performance reasons

```
public class DatabaseService {  
    public MyEntity getEntityById(int id) {  
        // go to database and fetch the entity  
        // return the entity  
    }  
}
```

```
public class MyDatabaseService extends DatabaseService {  
    private Map<Integer, MyEntity> entities = new HashMap<>();  
  
    @Override  
    public MyEntity getEntityById(int id) {  
        return entities.get(id);  
    }  
}
```

<https://www.youtube.com/watch?v=oFBkzrwwwW8>

STUBS

- A stub is an implementation that behaves "unnaturally".
- It is preconfigured (usually by the test set-up) to respond to **specific inputs** with **specific outputs**.
- The purpose of a stub is to get your system under test **into a specific state**.
- Examples: test whether SUT notifies users when a request to XXX REST API returns 404 error
 - You could stub out the REST API with an API that **always returns 404**

<https://stackoverflow.com/a/55030455/636398>

STUBS

- A stub is an implementation that behaves "unnaturally".
- It is preconfigured (usually by the test set-up) to respond to **specific inputs** with **specific outputs**.
- The purpose of a stub is to get your system under test **into a specific state**.

```
public class FacebookService {  
    public Profile getProfile(int profileId) throws Exception {  
        // calls Facebook's API  
        // retrieves the profile details  
        // returns the profile object  
    }  
}
```

```
public class MyFacebookService extends FacebookService {  
    public Profile getProfile(int profileId) throws Exception {  
        throw new ProfileNotFoundException();  
    }  
}
```

<https://www.youtube.com/watch?v=oFBkzrwwwW8>

MOCKS

- A mock is similar to a stub, but with **verification** added in.
- Purpose: make assertions about how your system under test interacted with the dependency (whether a function in SUT is called in the correct way).
- We use mocks when we don't want to invoke production code or when there is no easy way to verify that intended code was executed (e.g., sending emails)

```
public class MyUploadService extends UploadService {  
    @Override  
    public boolean uploadFile(String file) {  
        assert file.endsWith(".xml");  
        return true;  
    }  
}
```

<https://stackoverflow.com/a/55030455/636398>

- If you are writing a test for a system that uploads files to a website
- You could build a mock that accepts a file and assert that the uploaded file was correct.
- The mock doesn't actually upload the file (which is hard to verify)

MOCKING FRAMEWORKS

A mocking framework is a software library that makes it easier to create test doubles within tests; it allows you to replace an object with a mock, which is a test double whose behavior is specified inline in a test

Example 13-1. A credit card service

```
class PaymentProcessor {  
    private CreditCardService creditCardService;  
    ...  
    boolean makePayment(CreditCard creditCard, Money amount) {  
        if (creditCard.isExpired()) { return false; }  
        boolean success =  
            creditCardService.chargeCreditCard(creditCard, amount);  
        return success;  
    }  
}
```

```
class PaymentProcessorTest {  
    ...  
    PaymentProcessor paymentProcessor;  
  
    // Create a test double of CreditCardService with just one line of code.  
    @Mock CreditCardService mockCreditCardService;  
    @Before public void setUp() {  
        // Pass in the test double to the system under test.  
        paymentProcessor = new PaymentProcessor(mockCreditCardService);  
    }  
    @Test public void chargeCreditCardFails_returnFalse() {  
        // Give some behavior to the test double: it will return false  
        // anytime the chargeCreditCard() method is called. The usage of  
        // “any()” for the method’s arguments tells the test double to  
        // return false regardless of which arguments are passed.  
        when(mockCreditCardService.chargeCreditCard(any(), any()))  
            .thenReturn(false);  
        boolean success = paymentProcessor.makePayment(CREDIT_CARD, AMOUNT);  
        assertThat(success).isFalse();  
    }  
}
```

Mockito, a mocking framework for Java.

Example 13-1. A credit card service

```
class PaymentProcessor {  
    private CreditCardService creditCardService;  
    ...  
    boolean makePayment(CreditCard creditCard, Money amount) {  
        if (creditCard.isExpired()) { return false; }  
        boolean success =  
            creditCardService.chargeCreditCard(creditCard, amount);  
        return success;  
    }  
}
```

MOCKING FRAMEWORKS

Example 13-8. Stubbing

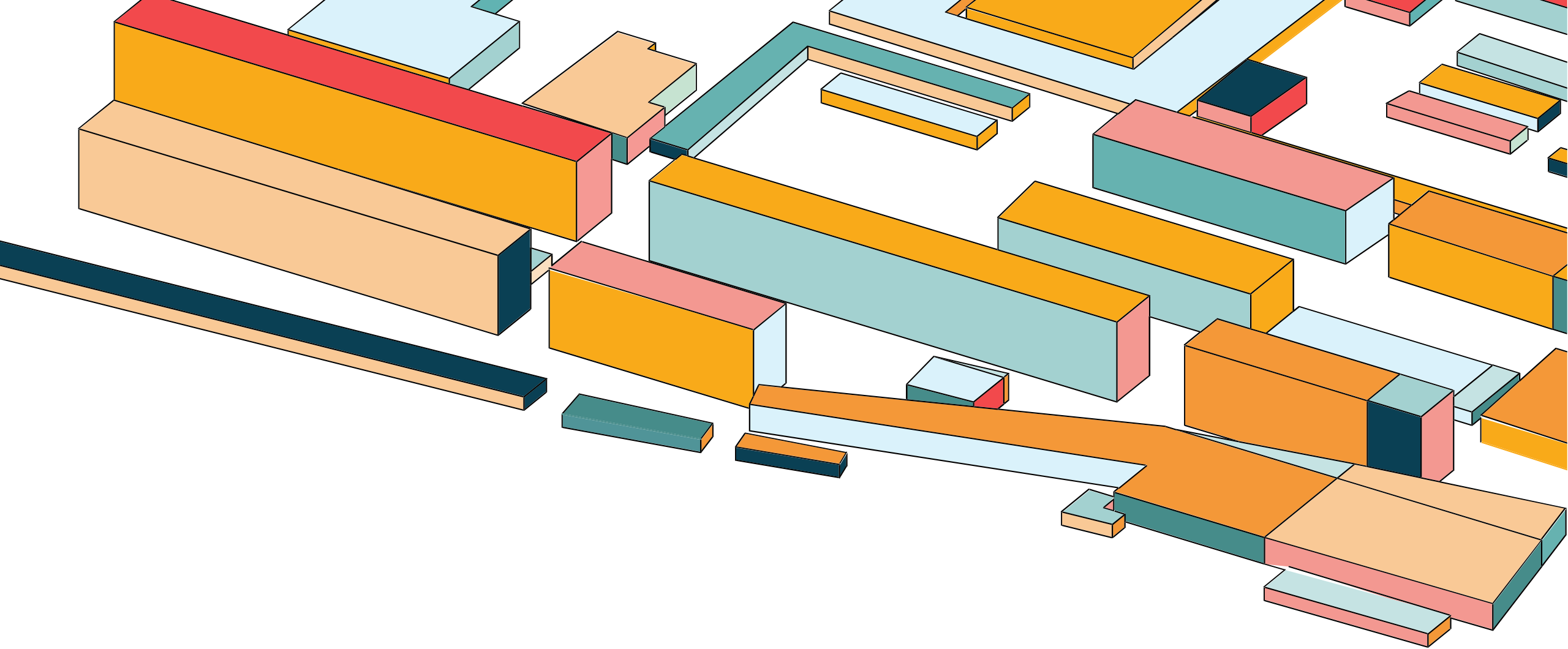
```
// Pass in a test double that was created by a mocking framework.
AccessManager accessManager = new AccessManager(mockAuthorizationService);

// The user ID shouldn't have access if null is returned.
when(mockAuthorizationService.lookupUser(USER_ID)).thenReturn(null);
assertThat(accessManager.userHasAccess(USER_ID)).isFalse();

// The user ID should have access if a non-null value is returned.
when(mockAuthorizationService.lookupUser(USER_ID)).thenReturn(USER);
assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

Example 13-1. A credit card service

```
class PaymentProcessor {
    private CreditCardService creditCardService;
    ...
    boolean makePayment(CreditCard creditCard, Money amount) {
        if (creditCard.isExpired()) { return false; }
        boolean success =
            creditCardService.chargeCreditCard(creditCard, amount);
        return success;
    }
}
```



MAINTAINABLE UNIT TESTS

UNCHANGING TESTS

- **Ideally**, after a test is written, it never needs to change unless the requirements of the system under test change
- Maintainable tests “just work”: after writing them, engineers don’t need to think about them again until they fail, and those failures indicate real bugs with clear causes.

UNCHANGING TESTS

After writing a test, you shouldn't need to touch that test again as you refactor the system, fix bugs, or add new features.

Refactorings: refactorings don't change the systems' behaviors, existing tests should remain unaffected

New features: new features may require new tests, but existing tests should remain unaffected

Bug fixes: like new features, may require new tests, but existing tests should remain unaffected

Only changing the system's existing behavior would require the updates to existing tests

TEST VIA PUBLIC APIS

- Public APIs are exposed to users
 - Hence, testing public APIs mimic how users use the system
-
- Public APIs change much less frequently than internal implementations
 - Hence, tests on public APIs change less frequently (more maintainable)

TEST VIA PUBLIC APIS

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + ", " + t.getRecipient() + ", " + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine the account balance
}
```

TEST VIA PUBLIC APIS

```
@Test
public void emptyAccountShouldNotBeValid() {
    assertThat(processor.isValid(newTransaction().setSender(EMPTY_ACCOUNT)))
        .isFalse();
}
```



Remove “private” modifier and test the internal implementation logic directly.

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + "," + t.getRecipient() + "," + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine the account balance
}
```

TEST VIA PUBLIC APIS

```
@Test
public void shouldSaveSerializedData() {
    processor.saveToDatabase(newTransaction()
        .setId(123)
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));
    assertThat(database.get(123)).isEqualTo("me, you, 100");
}
```



Test the internal states of the database.

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + ", " + t.getRecipient() + ", " + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine the account balance
}
```

TEST VIA PUBLIC APIS

- These tests are not maintainable
- Almost any refactoring of the system under test would cause the test to break, even if such a change would be invisible to the class's real users.
 - Renaming isValid()
 - Changing how String s is constructed in saveToDatabase()
 -

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + "," + t.getRecipient() + "," + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine the account balance
}
```

TEST VIA PUBLIC APIS

```
@Test
public void shouldTransferFunds() {
    processor.setAccountBalance("me", 150);
    processor.setAccountBalance("you", 20);

    processor.processTransaction(newTransaction()
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));

    assertThat(processor.getAccountBalance("me")).isEqualTo(50);
    assertThat(processor.getAccountBalance("you")).isEqualTo(120);
}
```



The test access the system in the same manner as the users would via public APIs

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + "," + t.getRecipient() + "," + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine the account balance
}
```


TEST VIA PUBLIC APIS

```
@Test
public void shouldTransferFunds() {
    processor.setAccountBalance("me", 150);
    processor.setAccountBalance("you", 20);

    processor.processTransaction(new Transaction()
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));

    assertThat(processor.getAccountBalance("me")).isEqualTo(50);
    assertThat(processor.getAccountBalance("you")).isEqualTo(120);
}
```

- Such tests are more realistic and less brittle/more maintainable
- Internal refactoring won't affect the tests

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + "," + t.getRecipient() + "," + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine the account balance
}
```

TEST BEHAVIORS, NOT METHODS

- Common patten: every production method has a corresponding @test method
- Convenient at first
- But as the method grow more complex, the test also grows in complexity and becomes less maintainable

TEST BEHAVIORS, NOT METHODS

```
public void displayTransactionResults(User user, Transaction transaction) {  
    ui.showMessage("You bought a " + transaction.getItemName());  
    if (user.getBalance() < LOW_BALANCE_THRESHOLD) {  
        ui.showMessage("Warning: your balance is low!");  
    }  
}
```

@Test

```
public void testDisplayTransactionResults() {  
    transactionProcessor.displayTransactionResults(  
        newUserWithBalance(  
            LOW_BALANCE_THRESHOLD.plus(dollars(2))),  
        new Transaction("Some Item", dollars(3)));  
  
    assertThat(ui.getText()).contains("You bought a Some Item");  
    assertThat(ui.getText()).contains("your balance is low");  
}
```

The method to be tested



A method-driven test: as the method under test becomes more complex and implements more functionality, its unit test will become increasingly convoluted and grow more and more difficult to work with.

TEST BEHAVIORS, NOT METHODS

```
public void displayTransactionResults(User user, Transaction transaction) {  
    ui.showMessage("You bought a " + transaction.getItemName());  
    if (user.getBalance() < LOW_BALANCE_THRESHOLD) {  
        ui.showMessage("Warning: your balance is low!");  
    }  
}
```

The method to be tested

```
@Test  
public void displayTransactionResults_showsItemName() {  
    transactionProcessor.displayTransactionResults(  
        new User(), new Transaction("Some Item"));  
    assertThat(ui.getText()).contains("You bought a Some Item");  
}
```

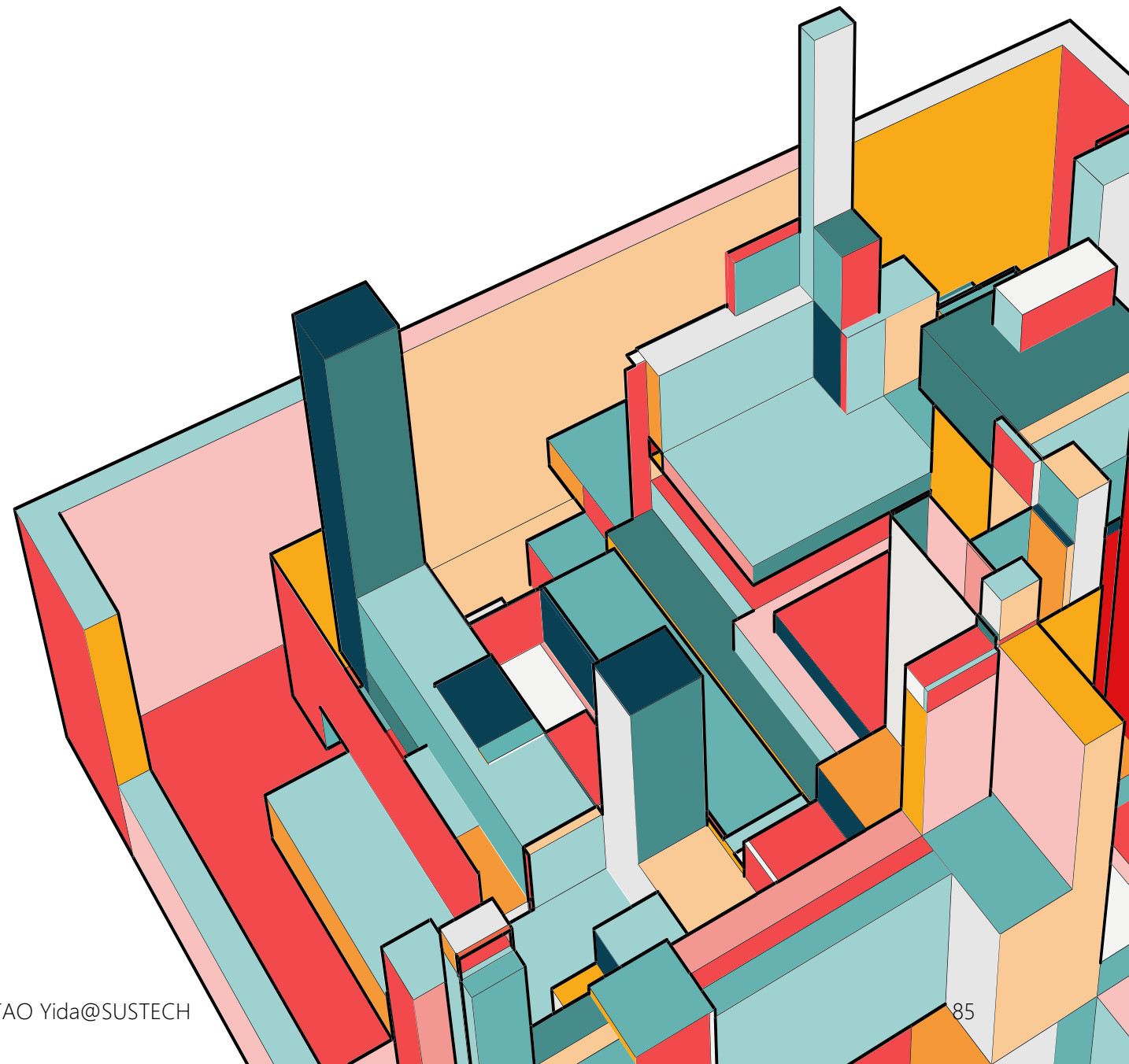


```
@Test  
public void displayTransactionResults_showsLowBalanceWarning() {  
    transactionProcessor.displayTransactionResults(  
        newUserWithBalance(  
            LOW_BALANCE_THRESHOLD.plus(dollars(2))),  
        new Transaction("Some Item", dollars(3)));  
    assertThat(ui.getText()).contains("your balance is low");  
}
```

A behavior-driven test: rather than writing a test for each method, write a test for each behavior.

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

- CI/CD
- Cloud native applications
- Deployment Pipelines