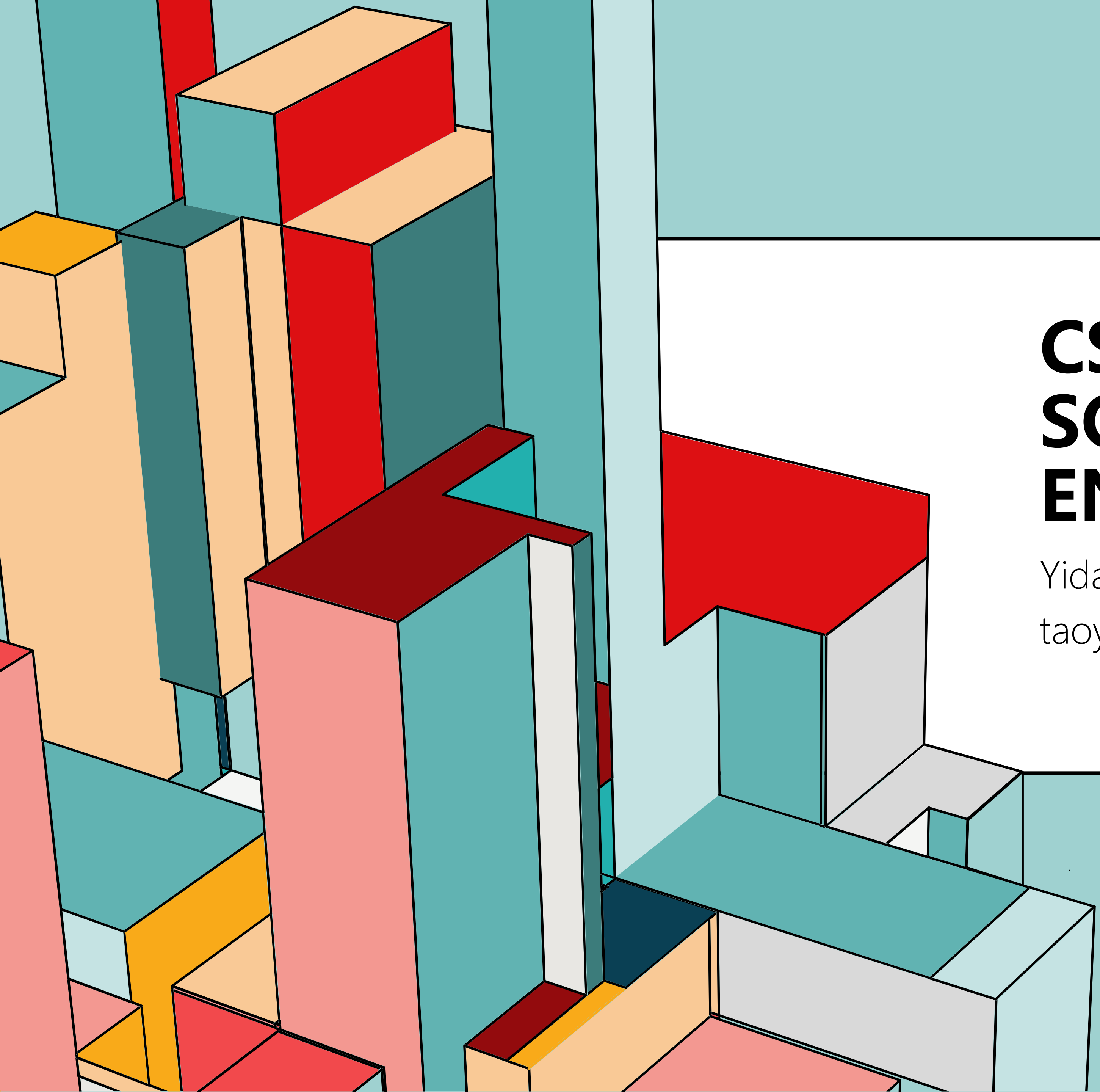
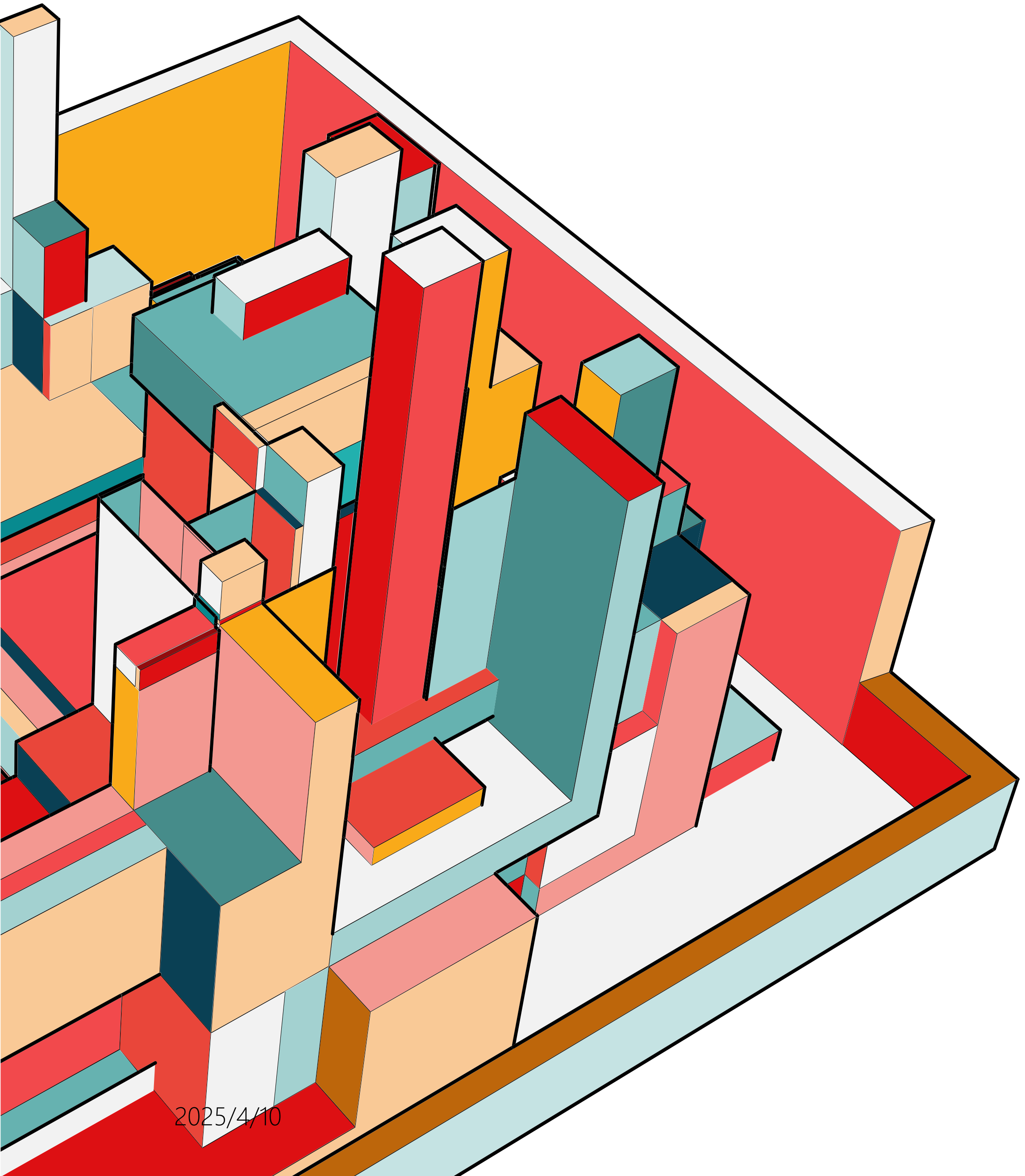


南方科技大學
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

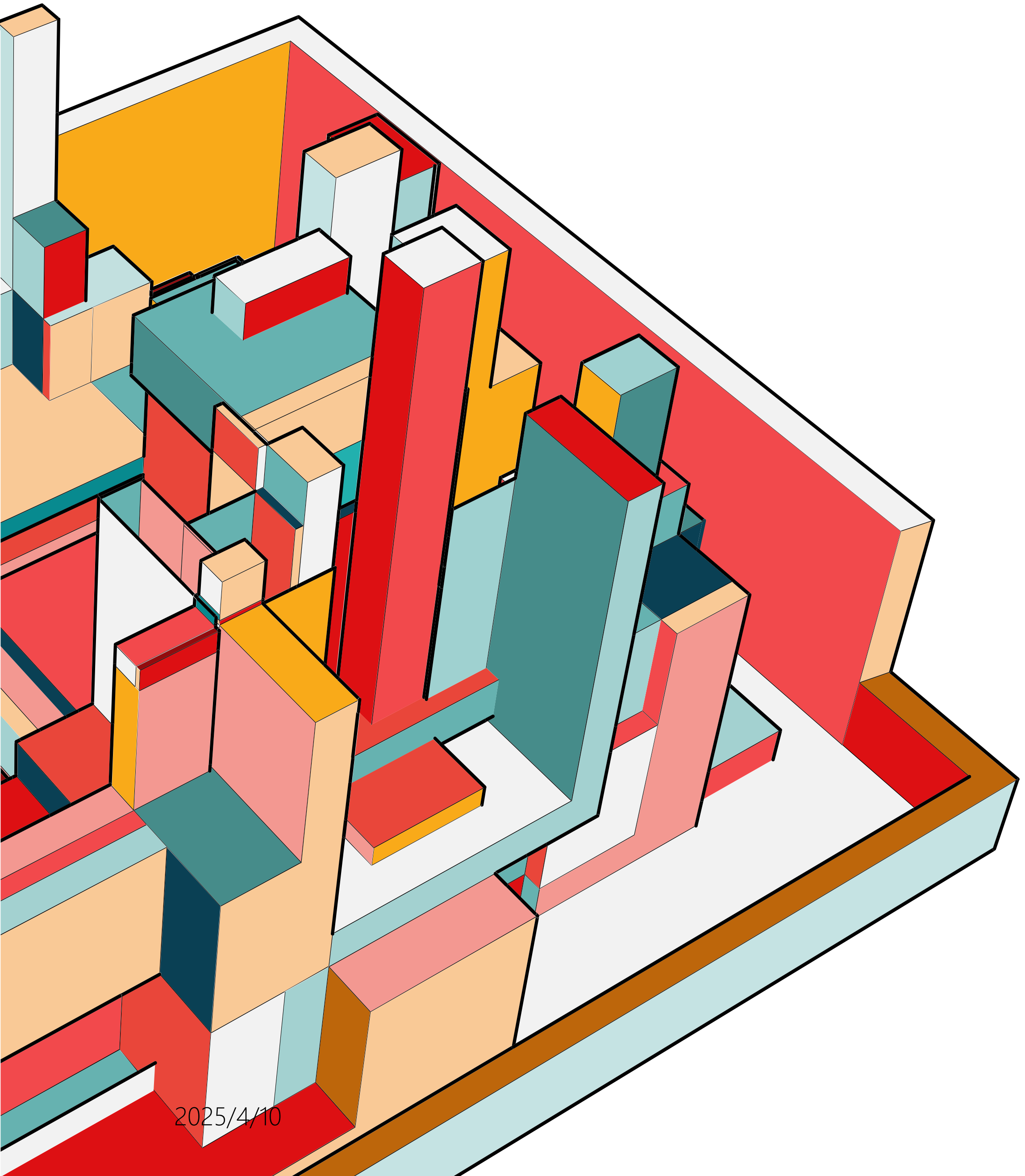
CS304 SOFTWARE ENGINEERING

Yida Tao
taoyd@sustech.edu.cn

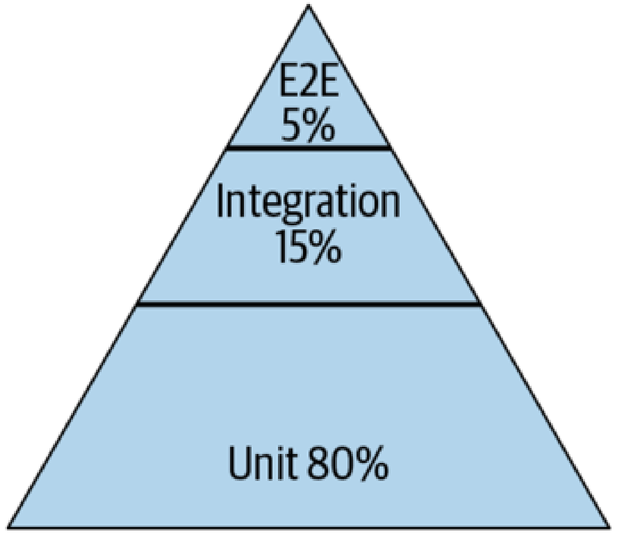


LECTURE 9

- Maintainable Unit Tests
- Integration tests: Test Doubles
- UI Testing



MAINTAINABLE UNIT TESTS



ULTIMATE GOAL: 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.

ULTIMATE GOAL: 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.

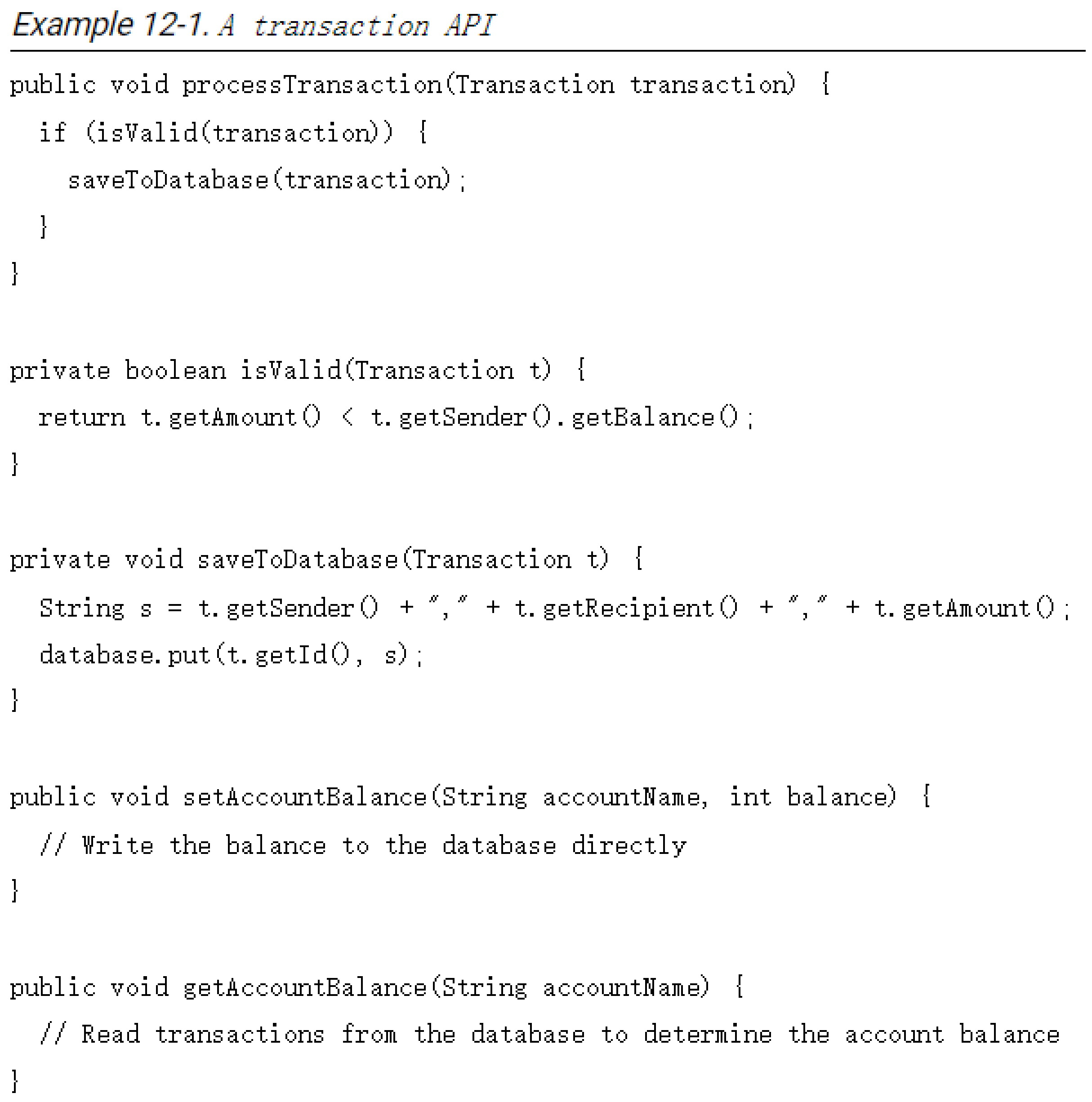
Refactorings: refactorings don't change the systems' behaviors, existing tests should remain unaffected
New features: new features may require new tests, but existing tests should remain unaffected
Bug fixes: like new features, may require new tests, but existing tests should remain unaffected

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

GOOD PRACTICE 1: TEST VIA PUBLIC APIS

- Write tests that invoke the SUT in the same way its users would
- That is, making calls to public APIs rather than private ones (implementation details)

BAD PRACTICE



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
}
```

BAD PRACTICE

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

Test the internal (private) implementation logic directly. This is NOT how real users would use the system

Even minor refactorings that are invisible to users can cause the test to break

- Renaming `isValid()`

Example 12-1. A transaction API

```java
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
}
```

BAD PRACTICE

```
@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.

Even minor refactorings that are invisible to users can cause the test to break

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

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

- Such tests are more realistic and maintainable
- Less brittle, as 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
}
```

GOOD PRACTICE 1: TEST VIA PUBLIC APIS

- Public APIs change much less frequently than internal implementations. Hence, tests on public APIs change less frequently (more maintainable)
- If tests work the same way as users, change that breaks a test may also break a user (i.e., test failures indicating real user-affecting problems)
- Bonus: such tests can serve as useful examples and documentation for users

GOOD PRACTICE 2: 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());



}
```

The method to be tested

```
@Test
public void testDisplayTransactionResults() {
  transactionProcessor.displayTransactionResults(
      newUser

      new Transaction("Some Item", dollars(3)));

  assertThat(ui.getText()).contains("You bought a Some Item");

}
```

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(
      newUser

      new Transaction("Some Item", dollars(3)));

  assertThat(ui.getText()).contains("You bought a Some Item");

}
```

The method to be tested is updated

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 is updated

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

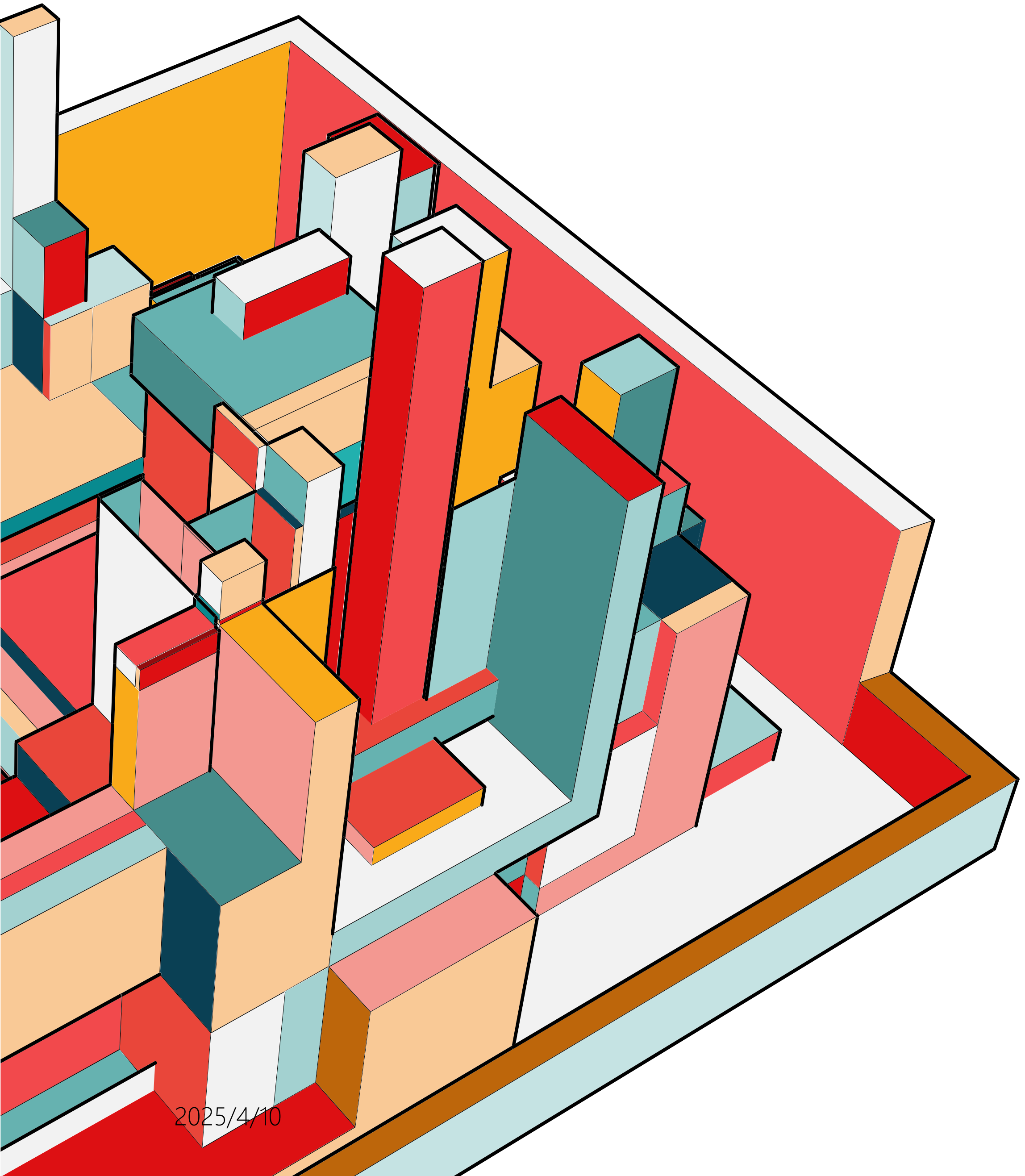
Behavior 1

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

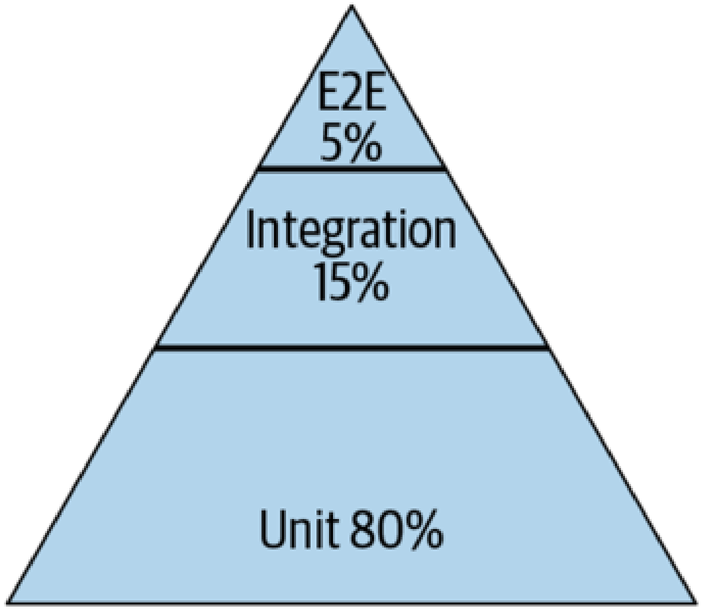
Behavior 2

```
@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");
}
```

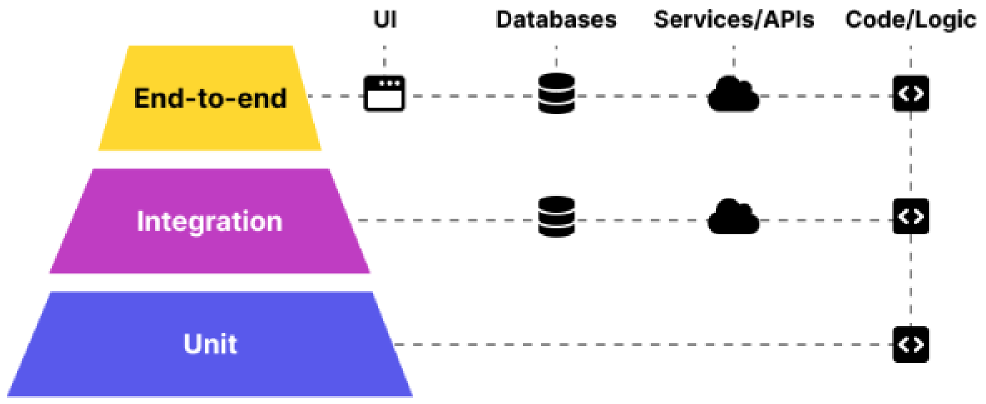
A behavior-driven test: rather than writing a test for each method, write a test for each behavior. √



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

WHAT ARE TEST DOUBLES?

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 replacement objects or code components that mimic the behavior of real objects or components in the SUT.
- Test doubles are commonly used to isolate the SUT and its dependencies, such as external services or libraries, by replacing them with lightweight, controllable substitutes.

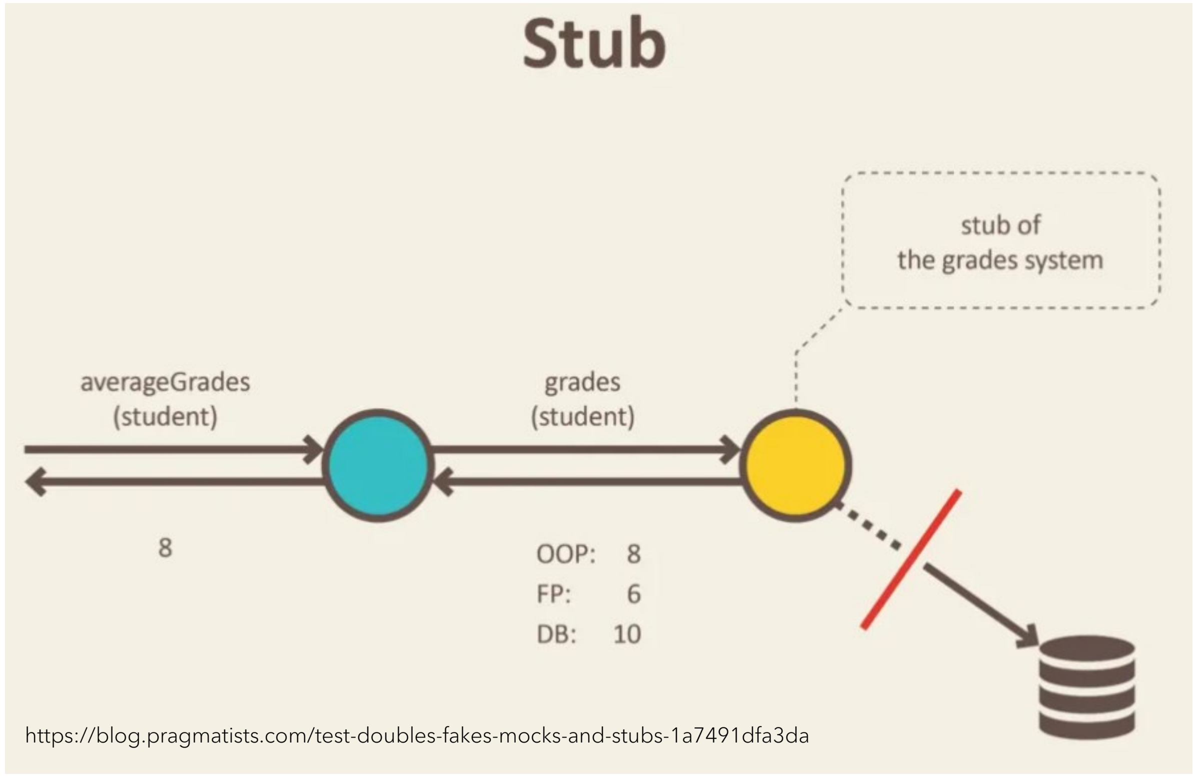
- Stubs
- Fakes
- Mocks

STUBS

- A stub returns predefined data to SUT
- A stub doesn’t contain real logic; it’s only there to respond to specific inputs with specific outputs.
- Stub is used when we cannot or don’t want to involve objects that would answer with real data or have undesirable side effects.

STUBS - EXAMPLES

- An object needs to grab some data from the database to respond to a method call.
- Instead of the real object from the database, we introduced a stub and defined what data should be returned.



https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

STUBS - EXAMPLES

Suppose we have a service that fetches user info:

```
public interface UserService {
    User getUserById(String userId);
}
```

The class under test:

```
public class UserProfile {
    private final UserService userService;

    public UserProfile(UserService userService) {
        this.userService = userService;
    }

    public String getUserEmail(String userId) {
        User user = userService.getUserById(userId);
        return user.getEmail();
    }
}
```

STUBS - EXAMPLES

Suppose we have a service that fetches user info:

```
public interface UserService {
    User getUserById(String userId);
}
```

We don't have user data, or we don't want to use real user data, we can create a stub with predefined data

```
public class StubUserService implements UserService {
    public User getUserById(String userId) {
        return new User("Alice", "alice@example.com");
    }
}
```

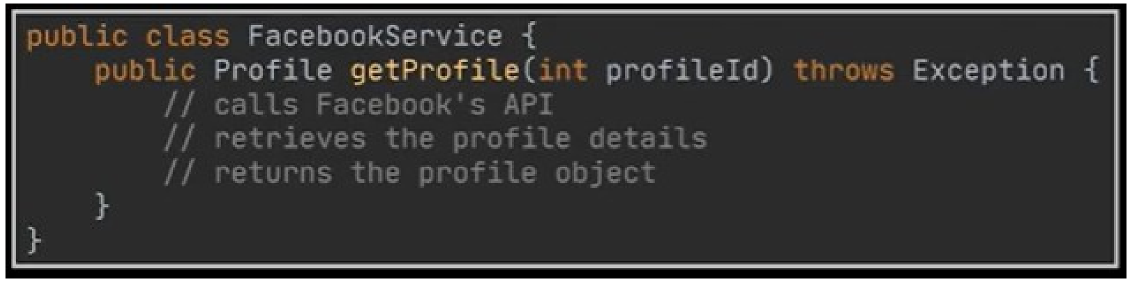
STUBS - EXAMPLES

```java
@Test
void testGetUserEmail() {
    UserService stub = new StubUserService();
    UserProfile profile = new UserProfile(stub);
    assertEquals("alice@example.com", profile.getUserEmail("123"));
}
```

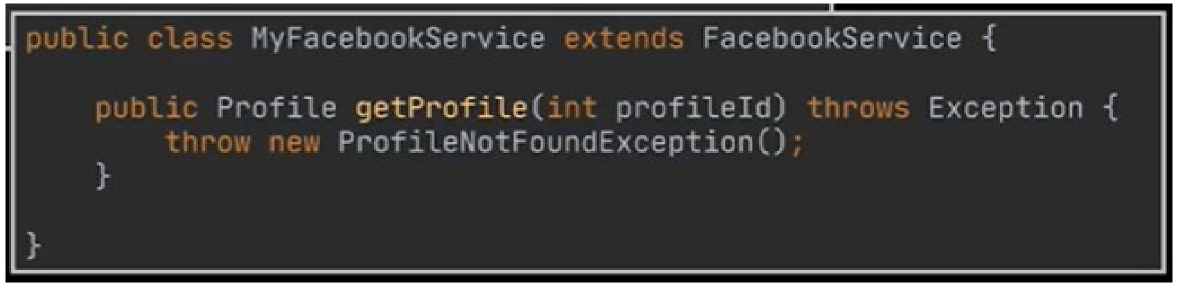
We can easily replace Stub with real object in later testing.

STUBS - 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**
- Test whether SUT notifies users when user profiles are not found
  - You could stub out the API that **always throws `ProfileNotFoundException`**



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



```java
public class MyFacebookService extends FacebookService {

    public Profile getProfile(int profileId) throws Exception {
        throw new ProfileNotFoundException();
    }

}
```

The purpose of a stub is to get your SUT **into a specific state**.

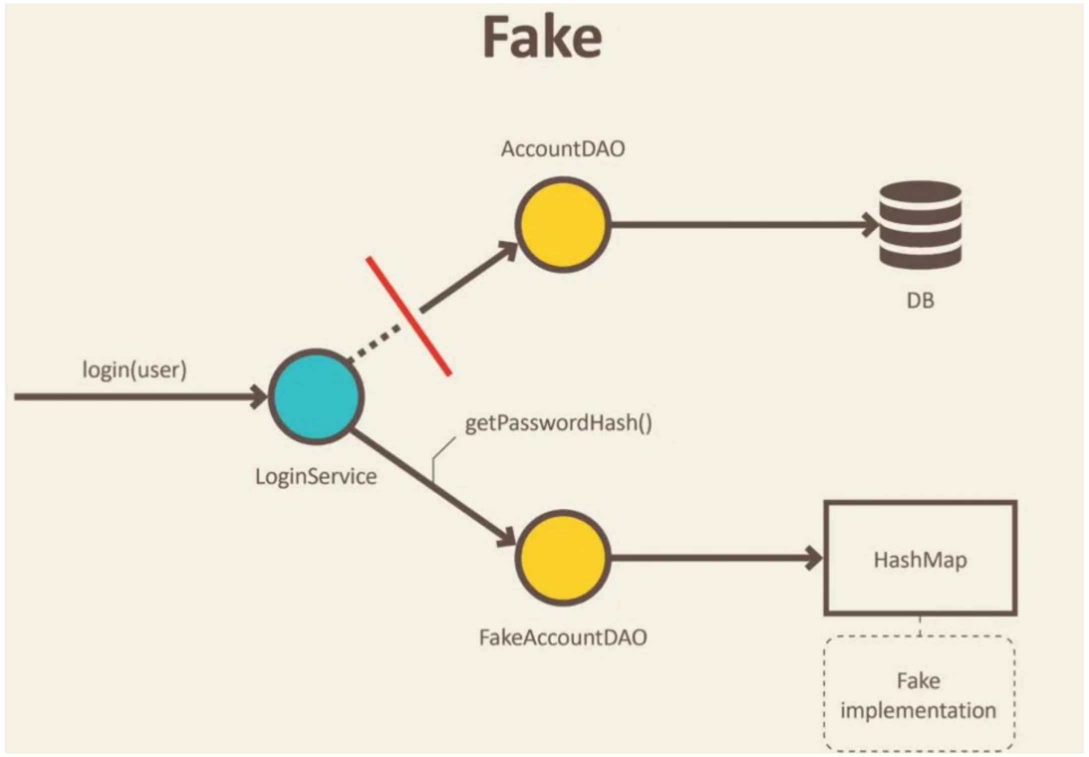
https://www.youtube.com/watch?v=oFBkzrwwwW8

FAKES

- A fake implements real logic but in a simplified or in-memory way
- 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 (e.g., database), usually used for performance reasons

FAKES - EXAMPLES

- In-memory database: designed to enable minimal response times by eliminating the need to access disks.
- 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.

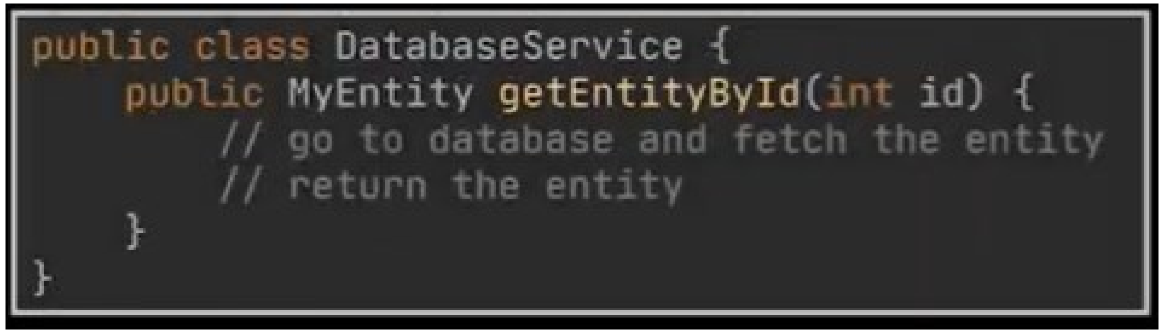


https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

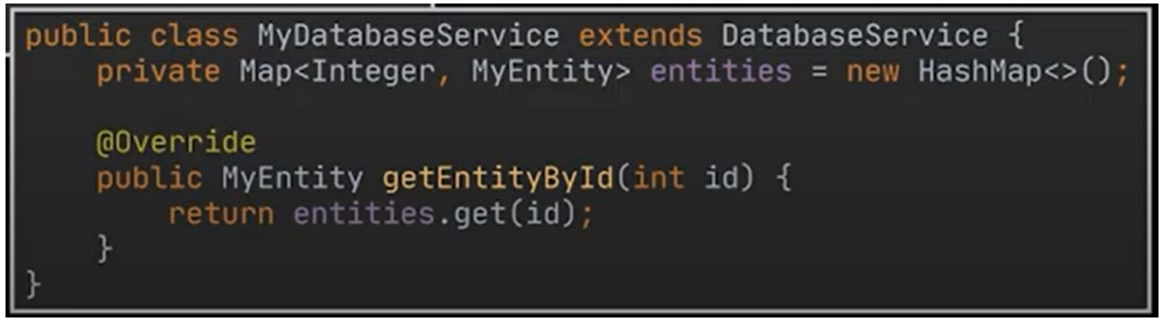
FAKES - EXAMPLES

- In-memory database: designed to enable minimal response times by eliminating the need to access disks.

- 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.



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



```java
public class MyDatabaseService extends DatabaseService {
    private Map<Integer, MyEntity> entities = new HashMap<>();

    @Override
    public MyEntity getEntityById(int id) {
        return entities.get(id);
    }
}
```

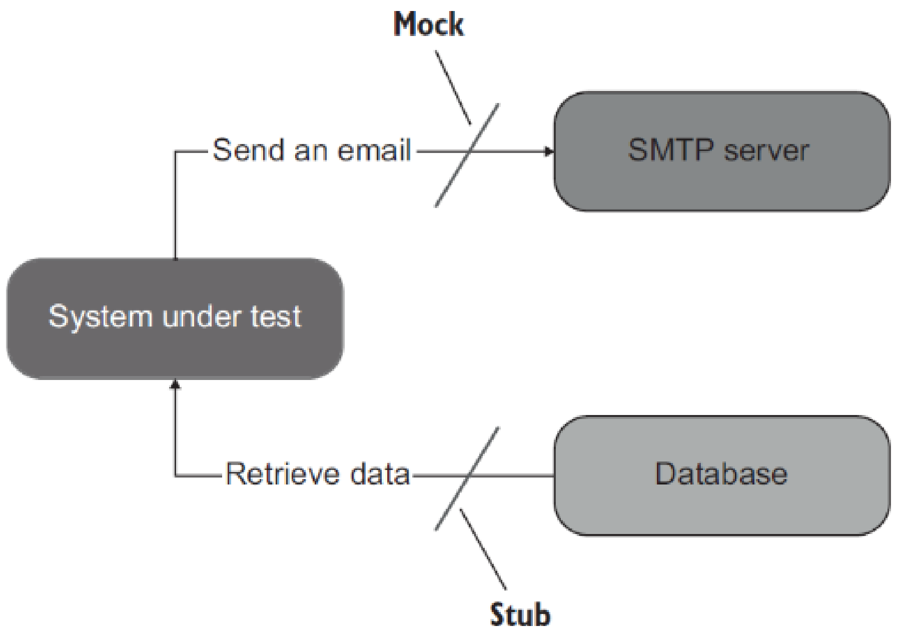
https://www.youtube.com/watch?v=oFBkzrwwwW8

MOCKS

- A mock is similar to a stub, but with **behavior verification** added in.
- Purpose: make assertions about how your SUT **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.

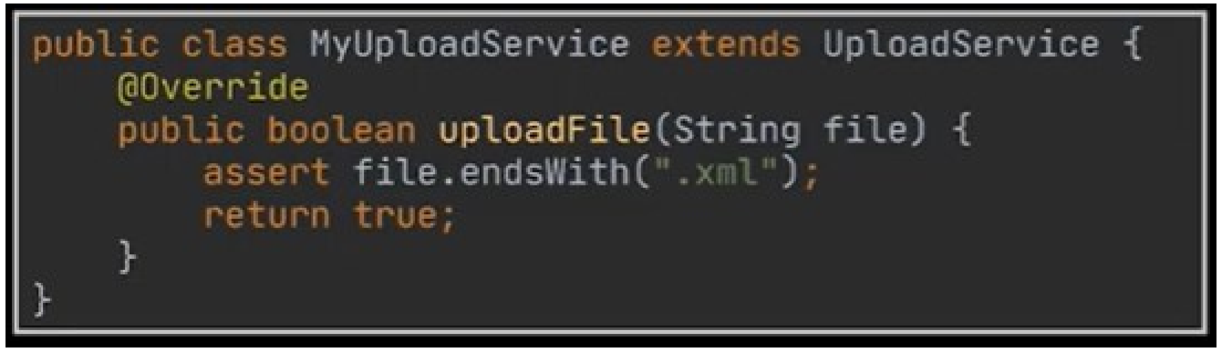
MOCKS - EXAMPLE 1

- Test whether SUT sends a welcome email when a user registers
  - We don’t want to send e-mails each time we run a test.
  - It’s also not easy to verify in tests that a right email was send.
  - Only thing we can do is to verify that e-mail sending service was called.



MOCKS - EXAMPLE 2

- 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 replaces the external file server, whose status is hard to verify



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

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

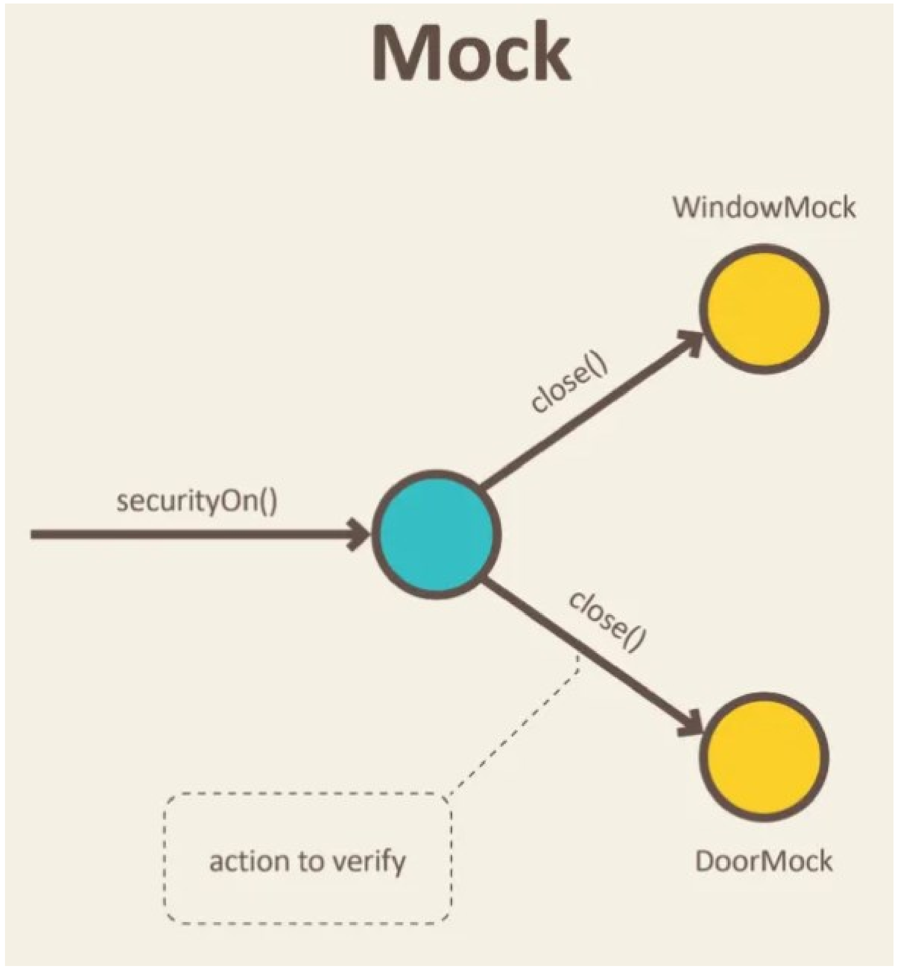
MOCKS - EXAMPLE 3

Example: a SecurityCentral software

- We don’t want to or can’t close real doors and windows to test
- Instead, we place door and window mocks objects in the test code.

Whether doors and windows will be closed for real are the responsibility of Doors and Windows, which should be tested in independent unit tests.

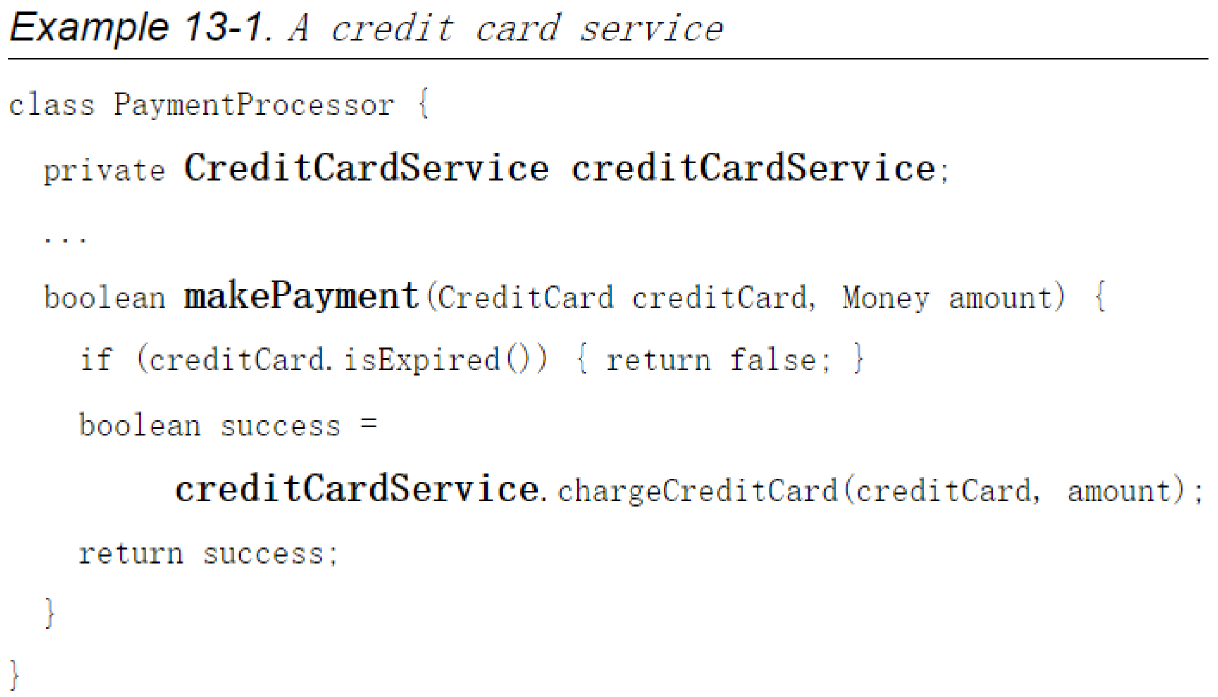
https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da



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;
  }
}
```

Mockito, a mocking framework for Java.

Example 13-6. Mocking frameworks

```
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();
  }
}
```

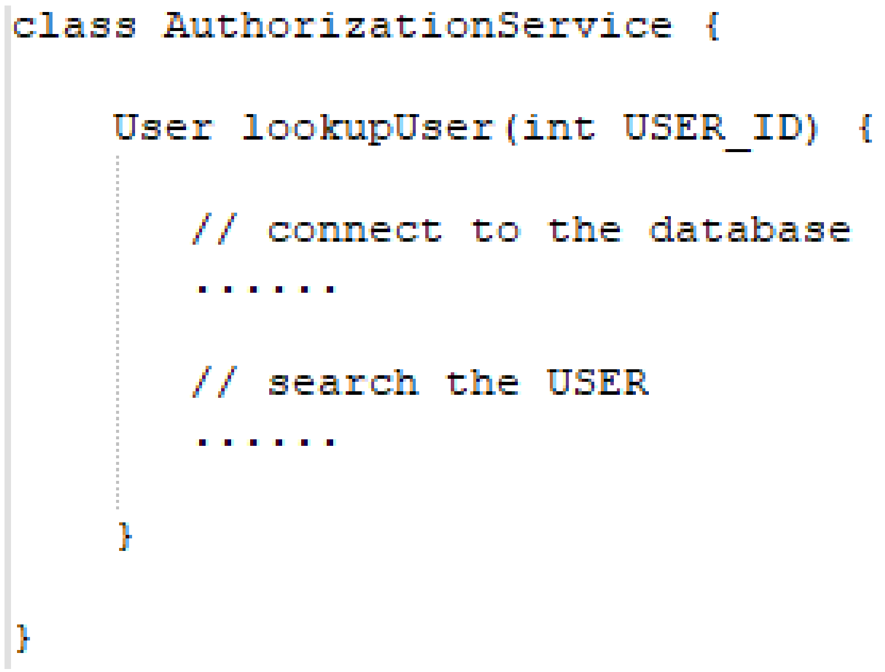
Example 13-1. A credit card service

```
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

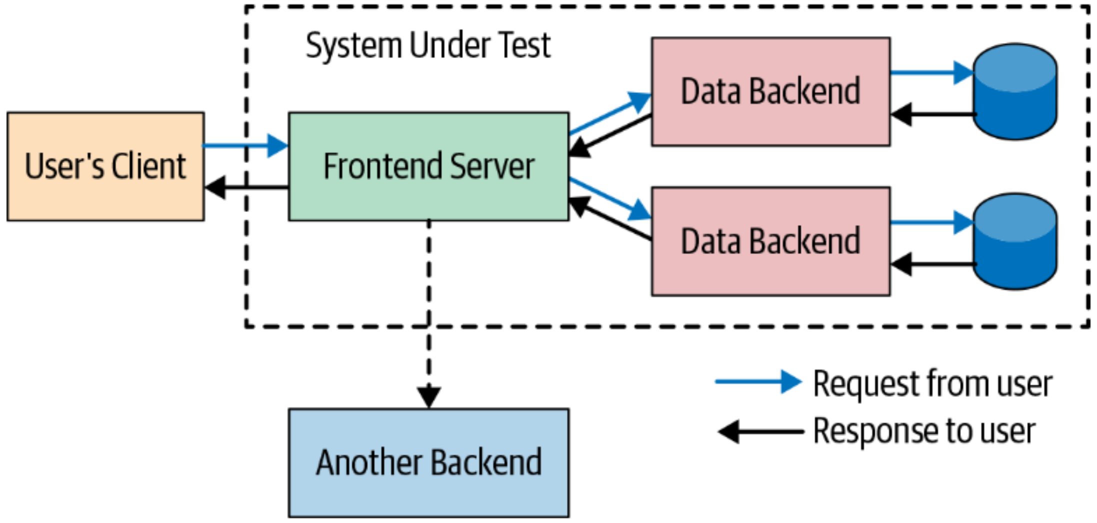
Example 13-8. Stubbing

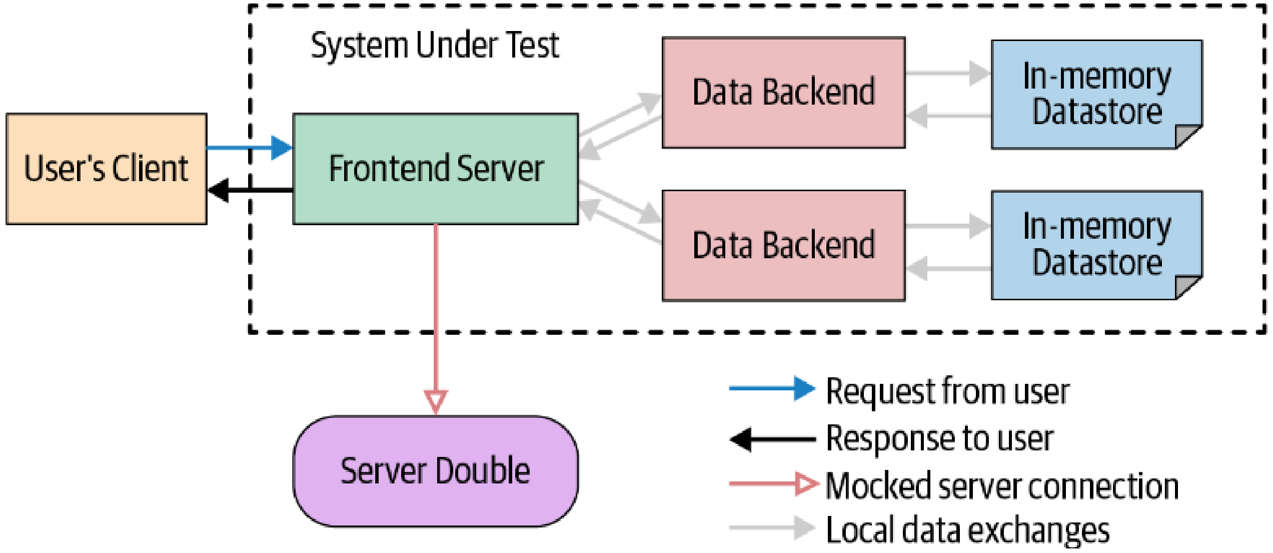
```
// Pass in a test double that was created by a mocking framework.
AccessManager accessManager = new AccessManager(mockAuthorizationService):

// 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();
```



```
class AuthorizationService {

    User lookupUser(int USER_ID) {

        // connect to the database
        ......

        // search the USER
        ......

    }

}
```

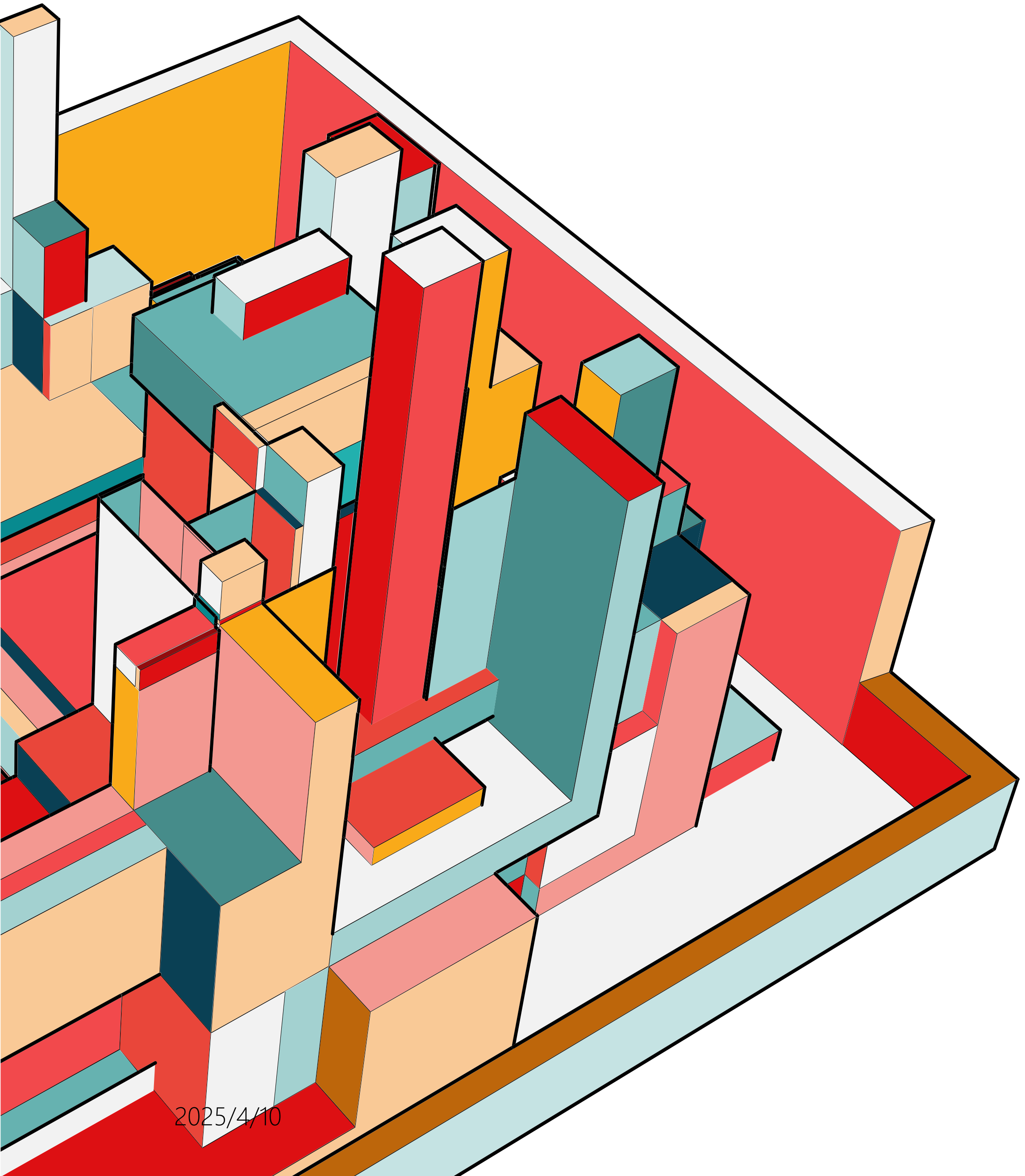
TEST DOUBLES SUMMARY

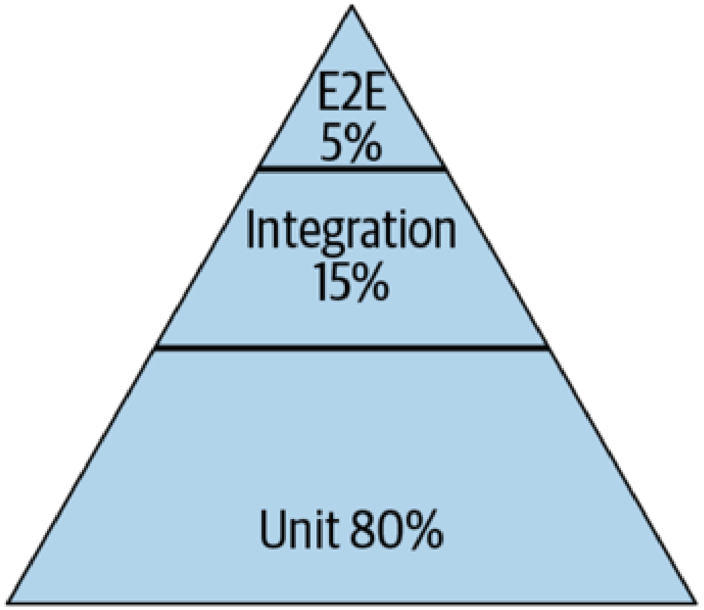




https://abseil.io/resources/swe-book/html/ch14.html

UI TESTING





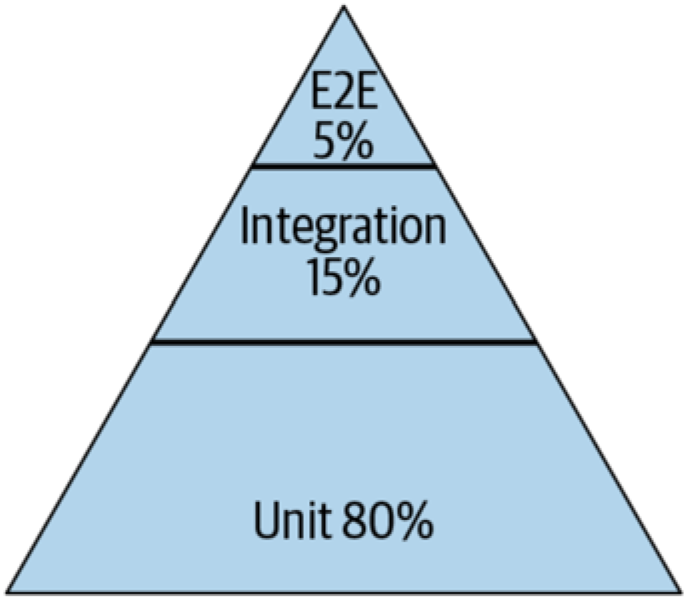
UI TESTING OVERVIEW

- UI is the visual part of a software that determines how users interact with the software and how information is displayed on the screen
- UI testing checks the application via the user interface, in the same way an end user would

UI TESTING OVERVIEW

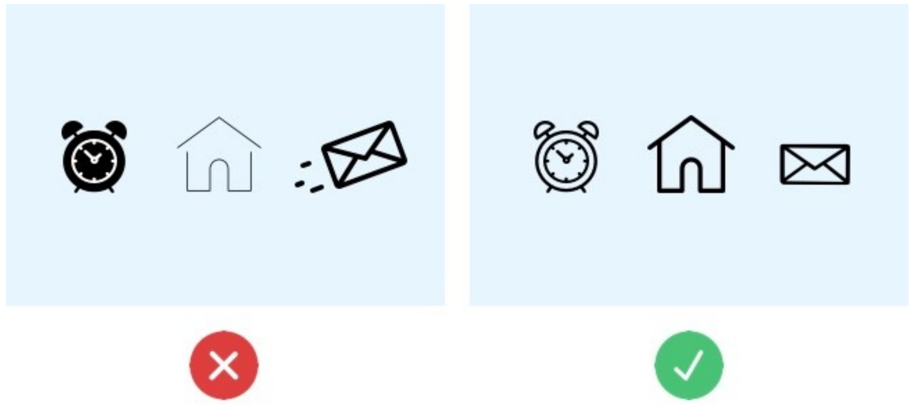
- UI tests are very powerful because they go “end-to-end” from the UI layer down, which test all different parts of an application from beginning (UI) to end (the underlying service, database, etc.)

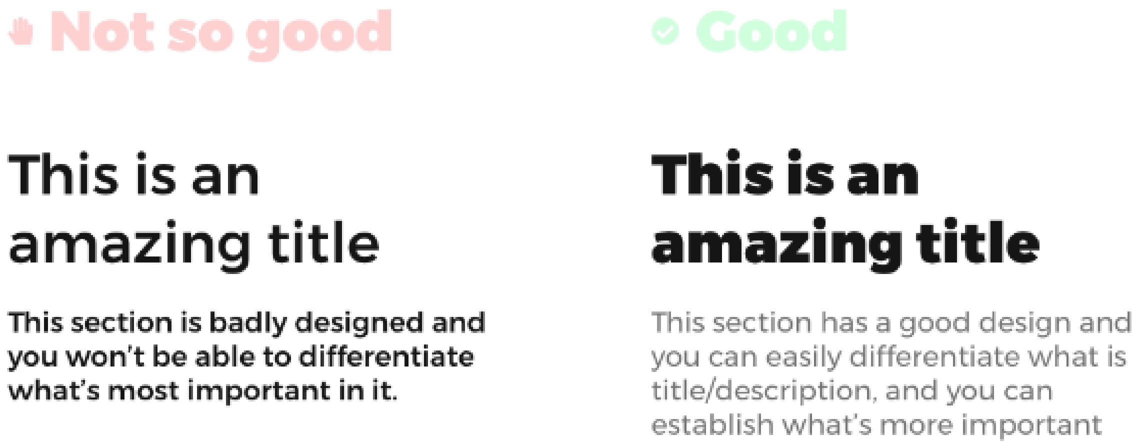
- UI testing can
    - Determine whether an application satisfies requirements
    - Demonstrate that it is fit for purpose
    - Detect defects



WHAT TO LOOK FOR IN UI TESTING?

- **Consistency**: consistency of colors, font types, and other visual elements.
- **Spelling**: spelling errors throughout the application
- **Typography**: Is the text easily readable? Does it have enough contrast with the background?



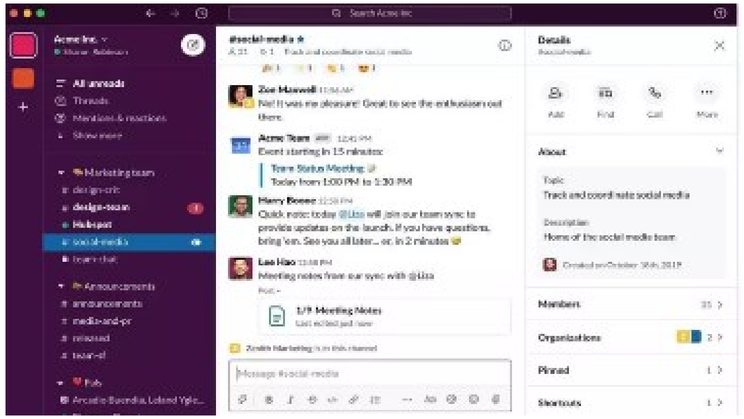

Not so good
This is an amazing title
This section is badly designed and you won't be able to differentiate what's most important in it.
Good
This is an amazing title
This section has a good design and you can easily differentiate what is title/description, and you can establish what's more important

Mostly manual

WHAT TO LOOK FOR IN UI TESTING?

- **Behavior of interactive elements**: Do the buttons, clicks, hovers, drag-and-drops, etc. work as intended?
- **Functional validation**: Do inputs produce the expected outputs?
- **Adaptability**: Ensure that UI elements adapt and are displayed correctly in different devices, browsers, screen formats and sizes, and OS.

Can be automated

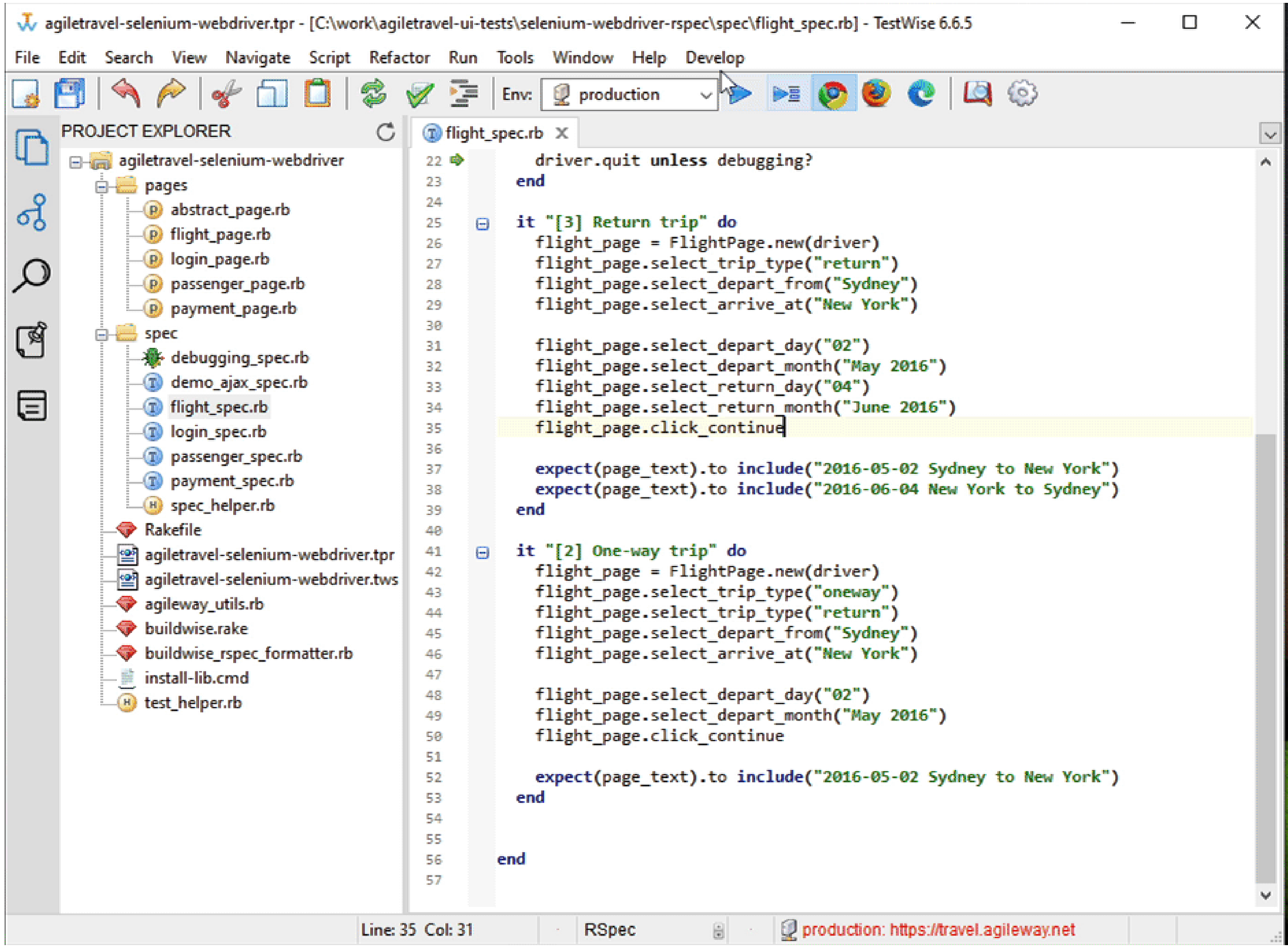


UI TESTING TOOLS

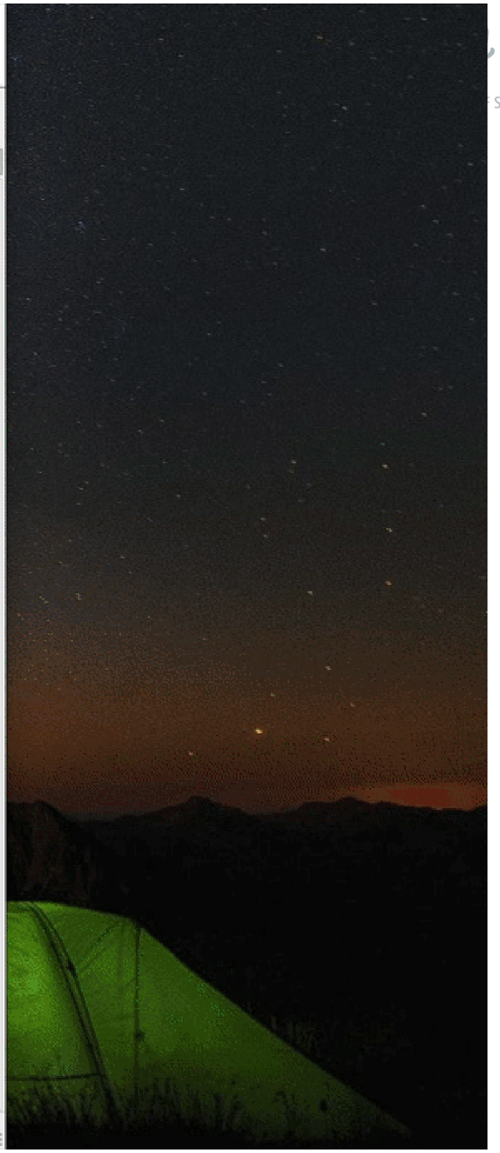
Example of UI functional validation:

- If you "add an item to cart", you need to validate that the correct item, price, and quantity are properly added to the cart.
- Is there validation in place to prevent users from entering unreasonable data such as negative numbers or special characters that aren't allowed?
- Can you verify that character length restrictions or required formats (e.g. phone number) aren't violated?

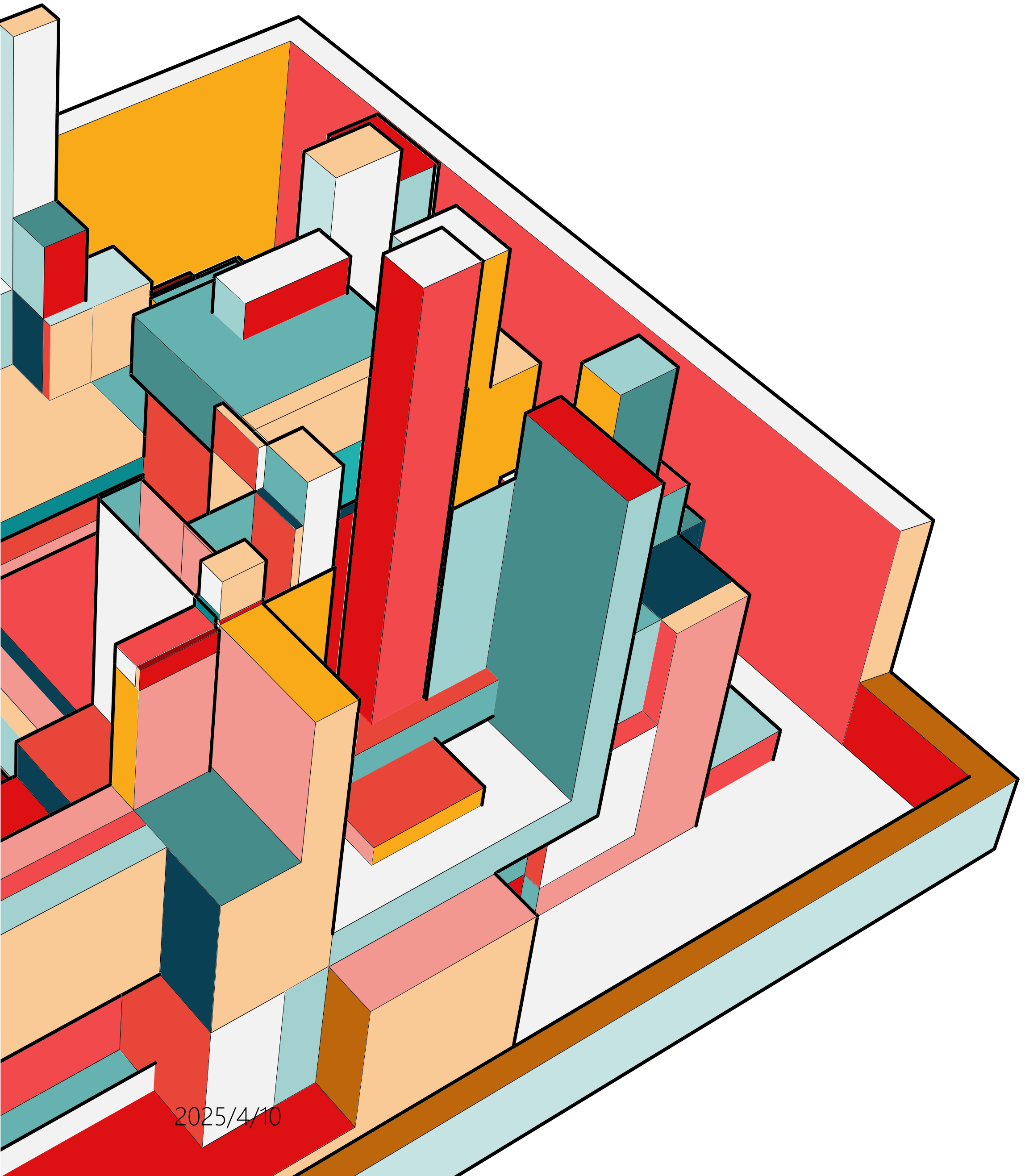
UI testing tools (e.g., Selenium) allow developers to create programs that act like robots, interacting with websites to check for any issues or bugs automatically.


agiletravel-selenium-webdriver.tpr - [C:\work\agiletravel-ui-tests\selenium-webdriver-rspec\spec\flight_spec.rb] - TestWise 6.6.5
File Edit Search View Navigate Script Refactor Run Tools Window Help Develop
Env: production
PROJECT EXPLORER
agiletravel-selenium-webdriver
pages
abstract_page.rb
flight_page.rb
login_page.rb
passenger_page.rb
payment_page.rb
spec
debugging_spec.rb
demo_ajax_spec.rb
flight_spec.rb
login_spec.rb
passenger_spec.rb
payment_spec.rb
spec_helper.rb
Rakefile
agiletravel-selenium-webdriver.tpr
agiletravel-selenium-webdriver.tws
agileway_utils.rb
buildwise.rake
buildwise_rspec_formatter.rb
install-lib.cmd
test_helper.rb
flight_spec.rb
Line: 35 Col: 31
RSpec
production: https://travel.agileway.net

```ruby
      driver.quit unless debugging?
    end

    it "[3] Return trip" do
      flight_page = FlightPage.new(driver)
      flight_page.select_trip_type("return")
      flight_page.select_depart_from("Sydney")
      flight_page.select_arrive_at("New York")

      flight_page.select_depart_day("02")
      flight_page.select_depart_month("May 2016")
      flight_page.select_return_day("04")
      flight_page.select_return_month("June 2016")
      flight_page.click_continue

      expect(page_text).to include("2016-05-02 Sydney to New York")
      expect(page_text).to include("2016-06-04 New York to Sydney")
    end

    it "[2] One-way trip" do
      flight_page = FlightPage.new(driver)
      flight_page.select_trip_type("oneway")
      flight_page.select_trip_type("return")
      flight_page.select_depart_from("Sydney")
      flight_page.select_arrive_at("New York")

      flight_page.select_depart_day("02")
      flight_page.select_depart_month("May 2016")
      flight_page.click_continue

      expect(page_text).to include("2016-05-02 Sydney to New York")
    end


  end
```



https://medium.com/geekculture/headless-browser-testing-clarified-d544ef0acf43



EVALUATING TEST CASES

How do we know that the tests are “good”?

- Test Metrics
- Mutation Testing
- Fuzzing

AIM OF COLLECTING TEST METRICS

Effectiveness

- How much of the software was tested?
- How good were the tests? Are we using low-value test cases?
- How many bugs did the test team (not) find?
- How bad are the bugs?
- How many bugs found were fixed? reopened? closed? deferred?

Efforts

- How long will it take to test?
- Will testing be done on time? Can the software be shipped on time?
- Was the test effort adequate? Could we have fit more testing in this release?
- How much money will it take to test?
- ……

COMMON TEST METRICS

- Coverage: statement, branch, condition coverage etc.

$$\text{Requirements Coverage} = \left(\frac{\textit{Number of requirements covered}}{\textit{Total number of requirements}}\right) \times 100$$

- Test tracking

$$\text{Passed Test Cases Percentage} = \left(\frac{\textit{Number of Passed Tests}}{\textit{Total number of tests executed}}\right) \times 100$$

$$\text{Failed Test Cases Percentage} = \left(\frac{\textit{Number of Failed Tests}}{\textit{Total number of tests executed}}\right) \times 100$$

COMMON TEST METRICS

- Defect tracking

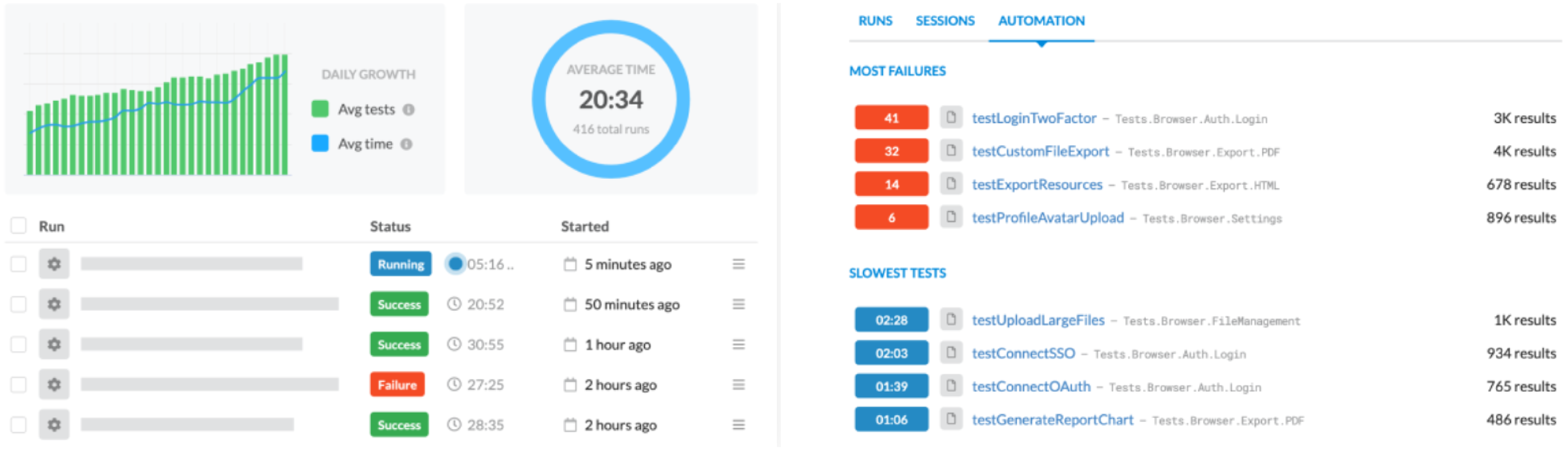
$$\text{Accepted Defects Percentage} = \left(\frac{\textit{Defects Accepted as Valid by Dev Team}}{\textit{Total defects reported}}\right) \times 100$$

$$\text{Critical Defects Percentage} = \left(\frac{\textit{Critical Defects}}{\textit{Total defects reported}}\right) \times 100$$

$$\text{Defects Rejected Percentage} = \left(\frac{\textit{Defects Rejected as invalid by Dev Team}}{\textit{Total defects reported}}\right) \times 100$$

$$\text{Defects Deferred Percentage} = \left(\frac{\textit{Defects deferred for future releases}}{\textit{Total defects reported}}\right) \times 100$$

COMMON TEST METRICS



Run	Status		Started
	Running	05:16..	5 minutes ago
	Success	20:52	50 minutes ago
	Success	30:55	1 hour ago
	Failure	27:25	2 hours ago
	Success	28:35	2 hours ago

MOST FAILURES

41	testLoginTwoFactor – Tests.Browser.Auth.Login	3K results
32	testCustomFileExport – Tests.Browser.Export.PDF	4K results
14	testExportResources – Tests.Browser.Export.HTML	678 results
6	testProfileAvatarUpload – Tests.Browser.Settings	896 results

SLOWEST TESTS

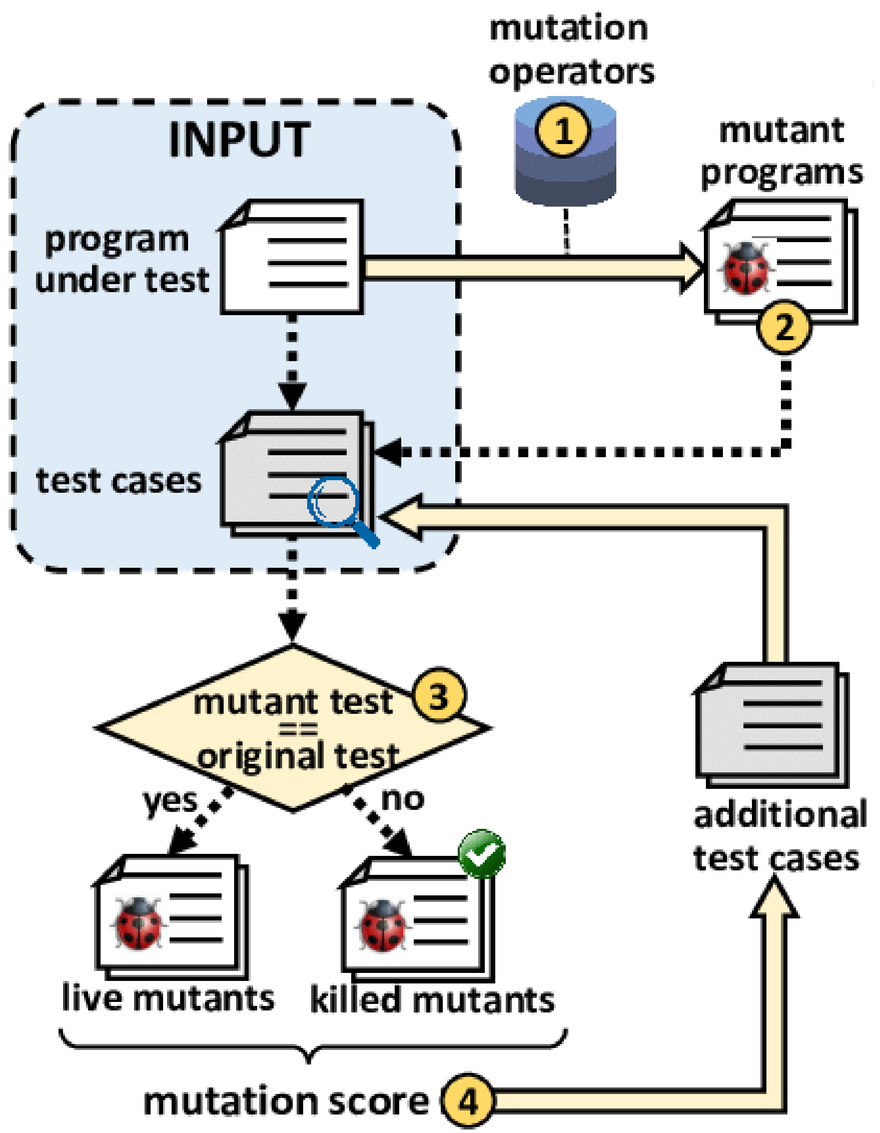
02:28	testUploadLargeFiles – Tests.Browser.FileManagement	1K results
02:03	testConnectSSO – Tests.Browser.Auth.Login	934 results
01:39	testConnectOAuth – Tests.Browser.Auth.Login	765 results
01:06	testGenerateReportChart – Tests.Browser.Export.PDF	486 results

https://www.testmo.com/qa-metrics-reporting

MUTATION TESTING

Are your tests sharp enough to catch small bugs?

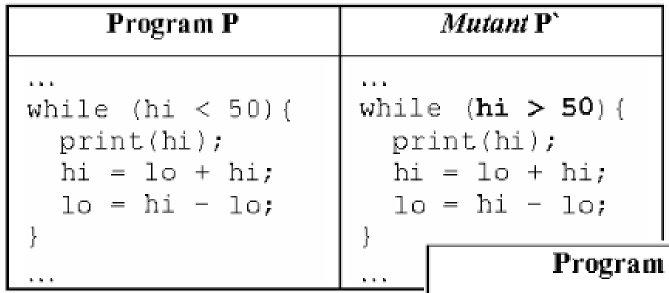
- The term **mutant** is borrowed from biology, where it refers to **a small variation of an original organism**.
- START: Mutation testing (变异测试) begins with a unit and a set of test cases for that unit, such that **all the test cases pass when executed.**



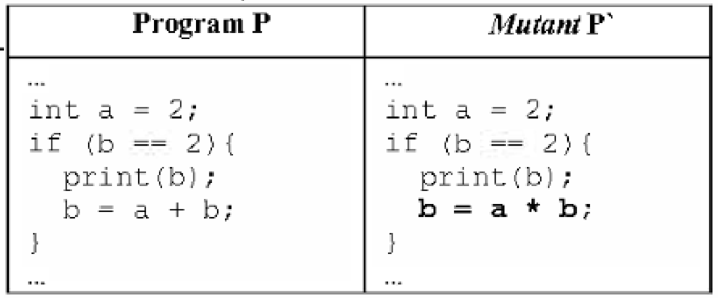
https://www.researchgate.net/figure/Mutation-testing-process_fig1_337518980

MUTATION TESTING

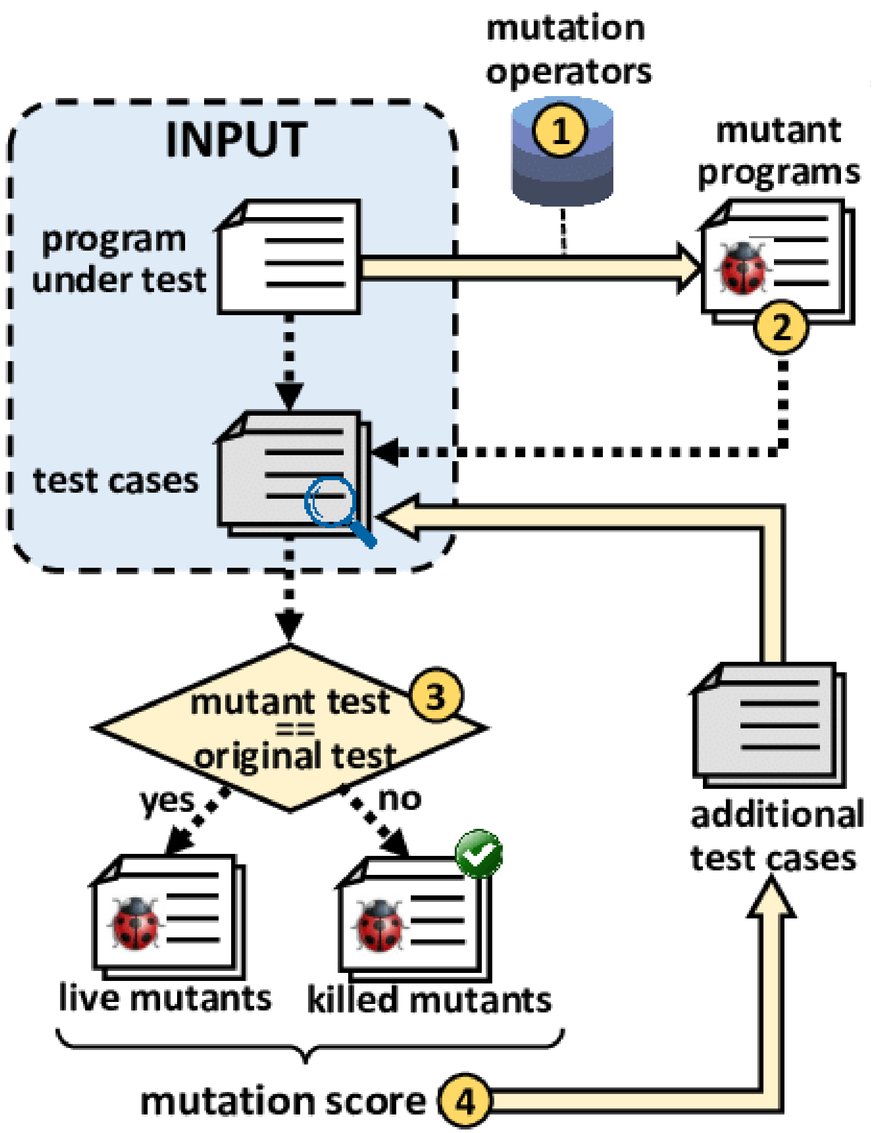
- STEP 1&2: A small change is made to the original unit, based on a list of predefined **mutation operators (突变算子)**.



Program P	*Mutant P`*
`...`	`...`
`while (hi < 50){`	`while (`**`hi > 50`**`){`
`  print(hi);`	`  print(hi);`
`  hi = lo + hi;`	`  hi = lo + hi;`
`  lo = hi - lo;`	`  lo = hi - lo;`
`}`	`}`
`...`	`...`



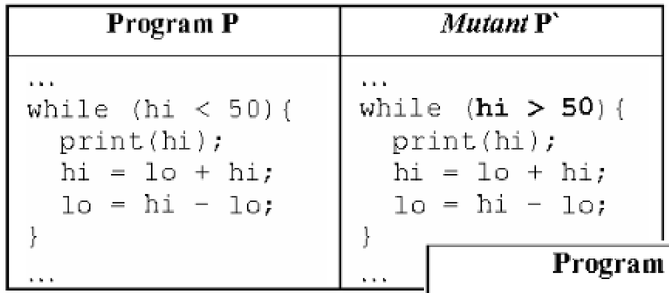
Program P	*Mutant P`*
`...`	`...`
`int a = 2;`	`int a = 2;`
`if (b == 2){`	`if (b == 2){`
`  print(b);`	`  print(b);`
`  b = a + b;`	`  `**`b = a * b;`**
`}`	`}`
`...`	`...`



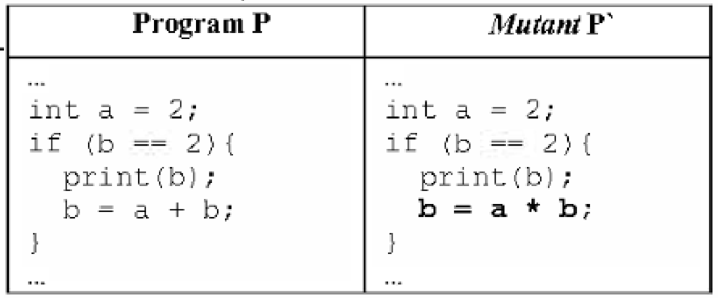
https://www.researchgate.net/figure/Mutation-testing-process_fig1_337518980

MUTATION TESTING

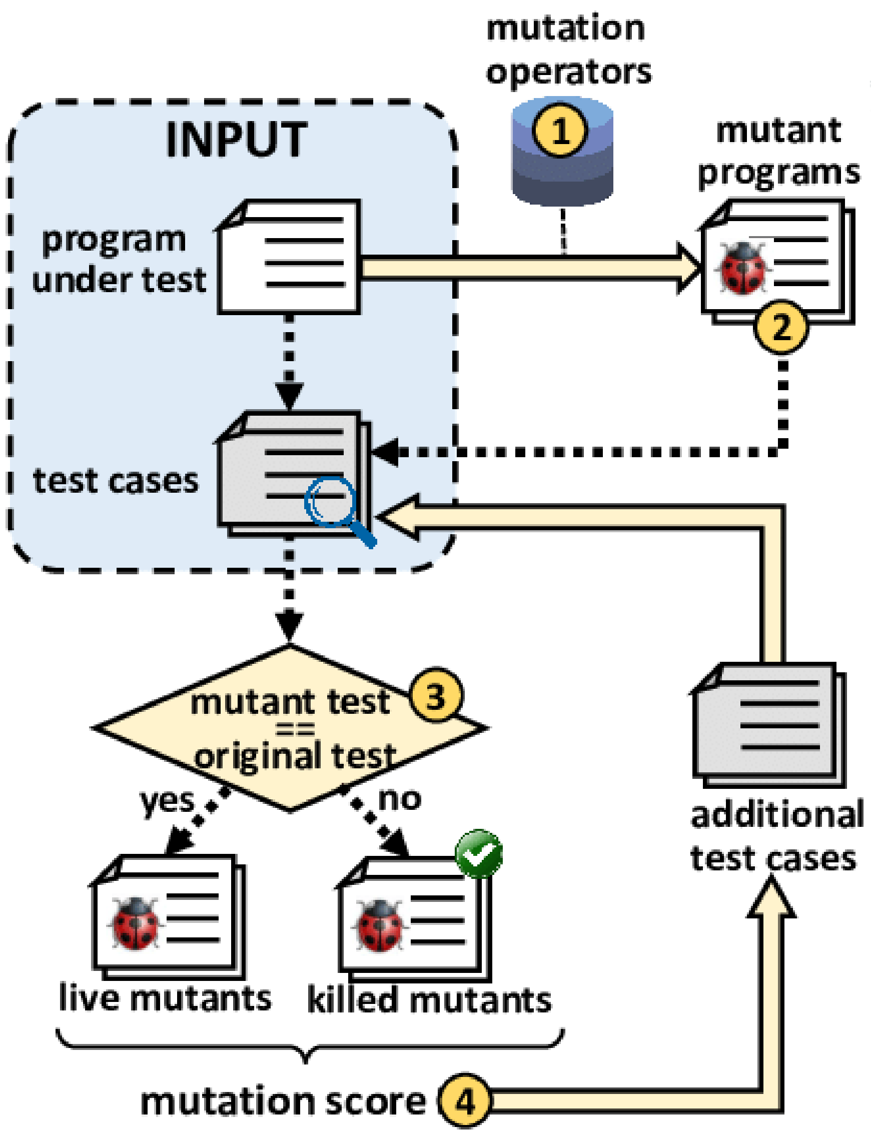
- STEP 3: tests are executed on the mutants.
- If all test cases succeed, the mutant lives.
- If at least one test case fails, the mutant is killed.



Program P	*Mutant* P`
`...`	`...`
`while (hi < 50){`	`while (`**`hi > 50`**`){`
`  print(hi);`	`  print(hi);`
`  hi = lo + hi;`	`  hi = lo + hi;`
`  lo = hi - lo;`	`  lo = hi - lo;`
`}`	`}`
`...`	`...`



Program P	*Mutant* P`
`...`	`...`
`int a = 2;`	`int a = 2;`
`if (b == 2){`	`if (b == 2){`
`  print(b);`	`  print(b);`
`  b = a + b;`	`  `**`b = a * b;`**
`}`	`}`
`...`	`...`



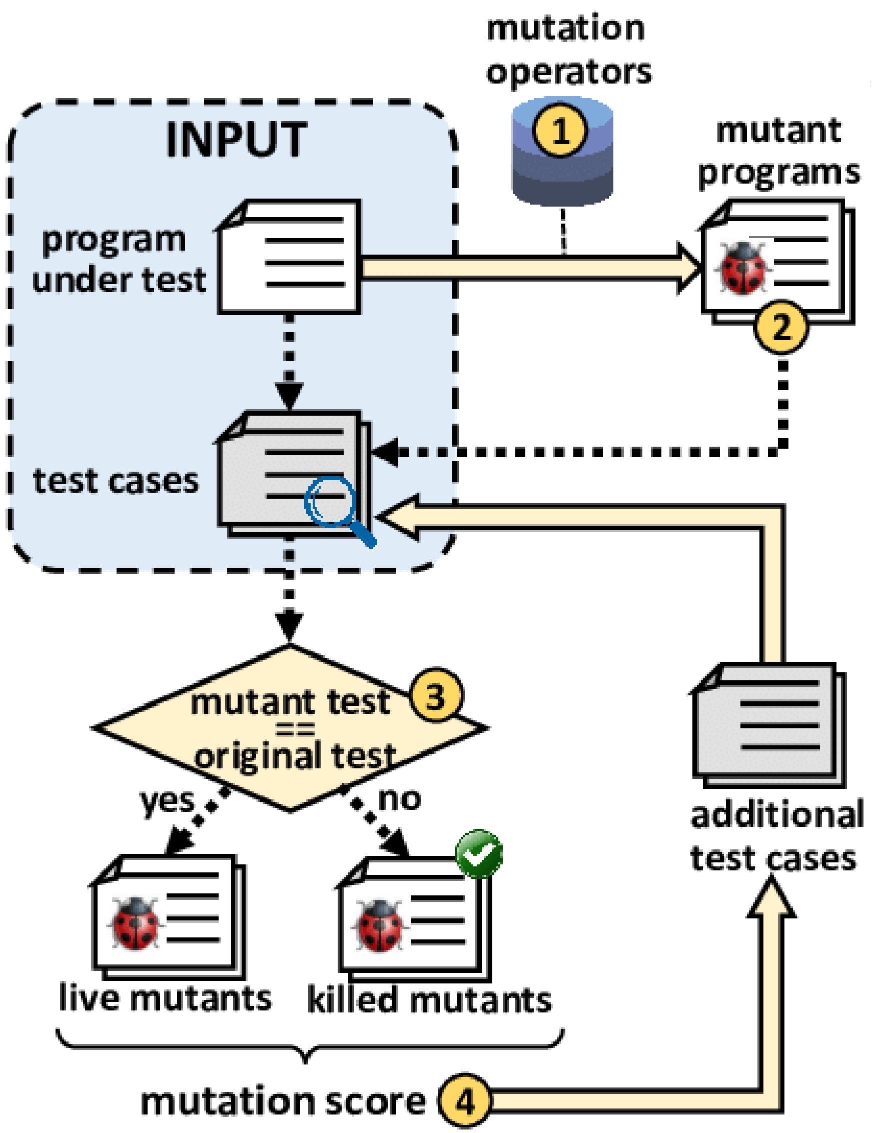
https://www.researchgate.net/figure/Mutation-testing-process_fig1_337518980

MUTATION TESTING

- STEP 4: Compute the mutation score

$$\text{Mutation score} = \frac{\#\ of\ killed\ mutants}{\#\ total\ mutants}$$

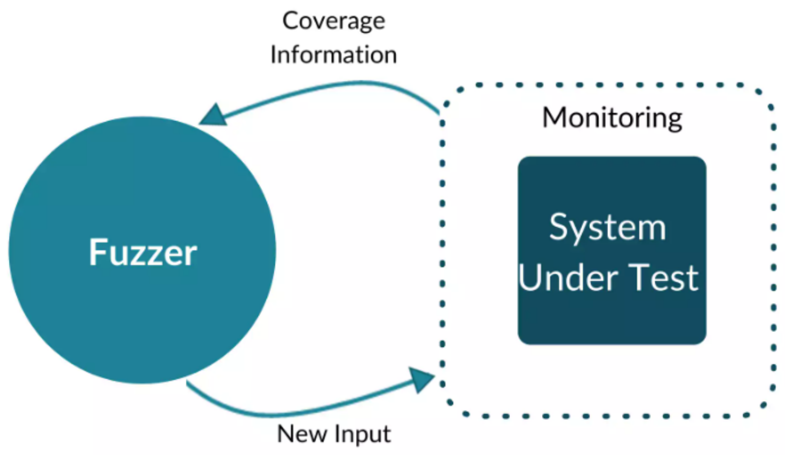
Higher mutation score increases the confidence of using the original test cases



https://www.researchgate.net/figure/Mutation-testing-process_fig1_337518980

FUZZING

- Fuzzing (模糊测试): an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to SUT, aiming to cause undesired behaviors such as exceptions, crashes, memory leaks, etc.
- The most commonly used fuzzers are **coverage-guided fuzzers**, which focus on finding inputs that cause new code coverage to explore edge cases that may crash the program.

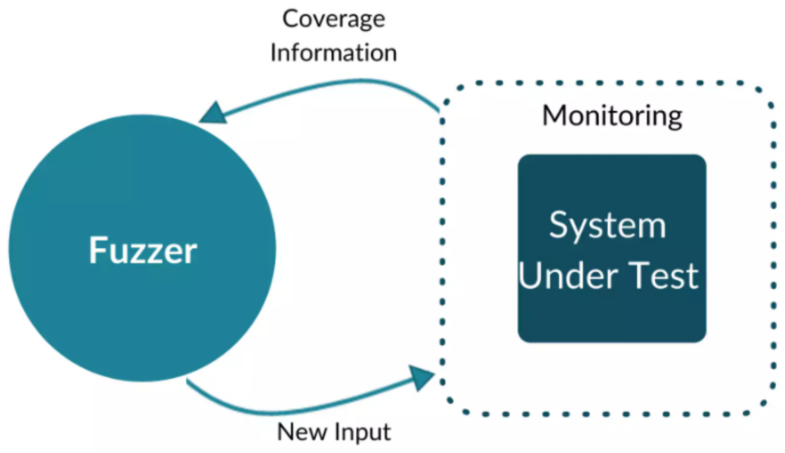


https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing

FUZZING

- Fuzzing (模糊测试): an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to SUT, aiming to cause undesired behaviors such as exceptions, crashes, memory leaks, etc.
- The most commonly used fuzzers are **coverage-guided fuzzers**, which focus on finding inputs that cause new code coverage to explore edge cases that may crash the program.

- null
- “”
- "; DROP TABLE users;"
- huge strings
- emojis or binary data

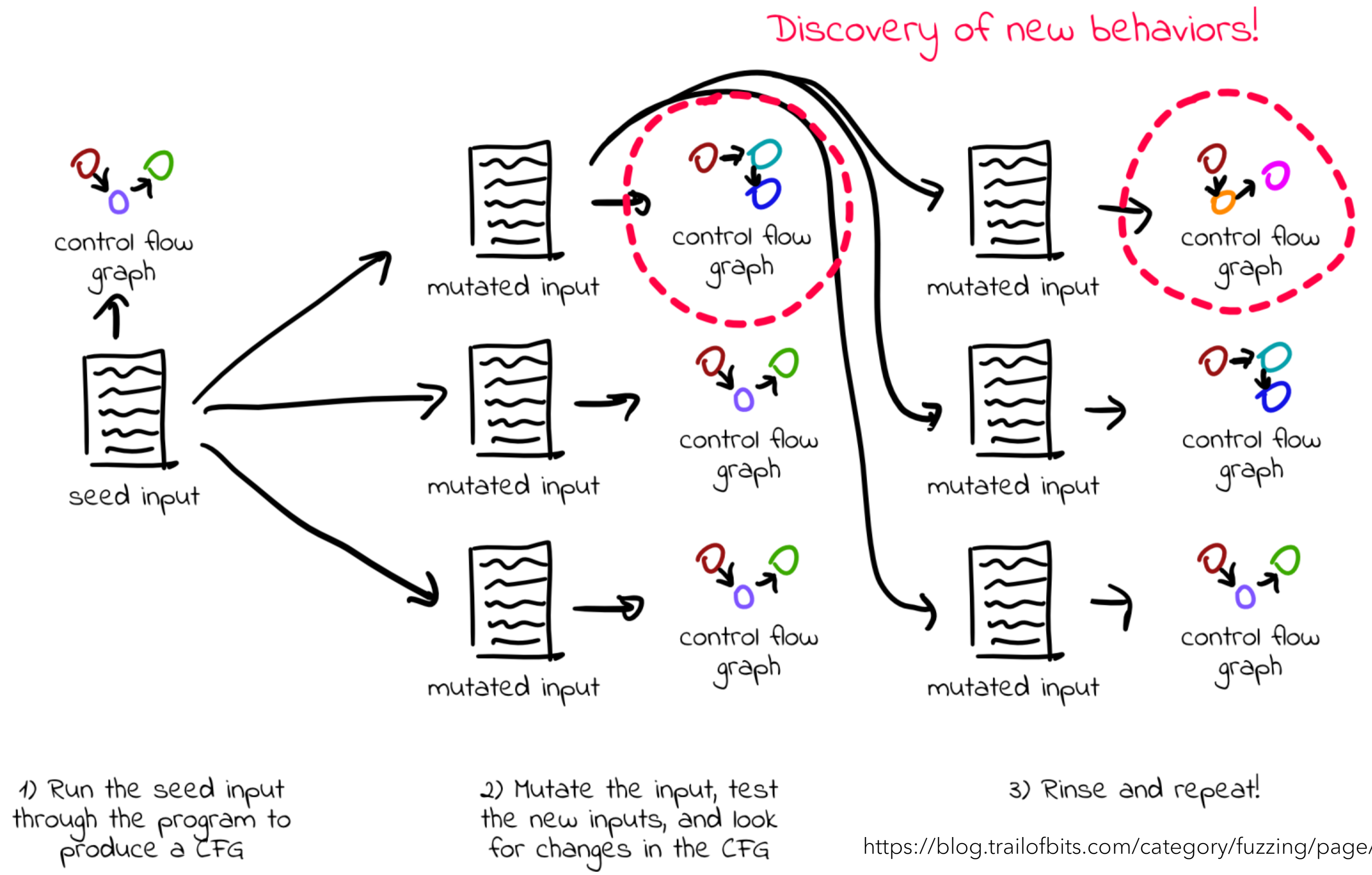


https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing

String parseUserInput(String input) { ... }

FUZZING

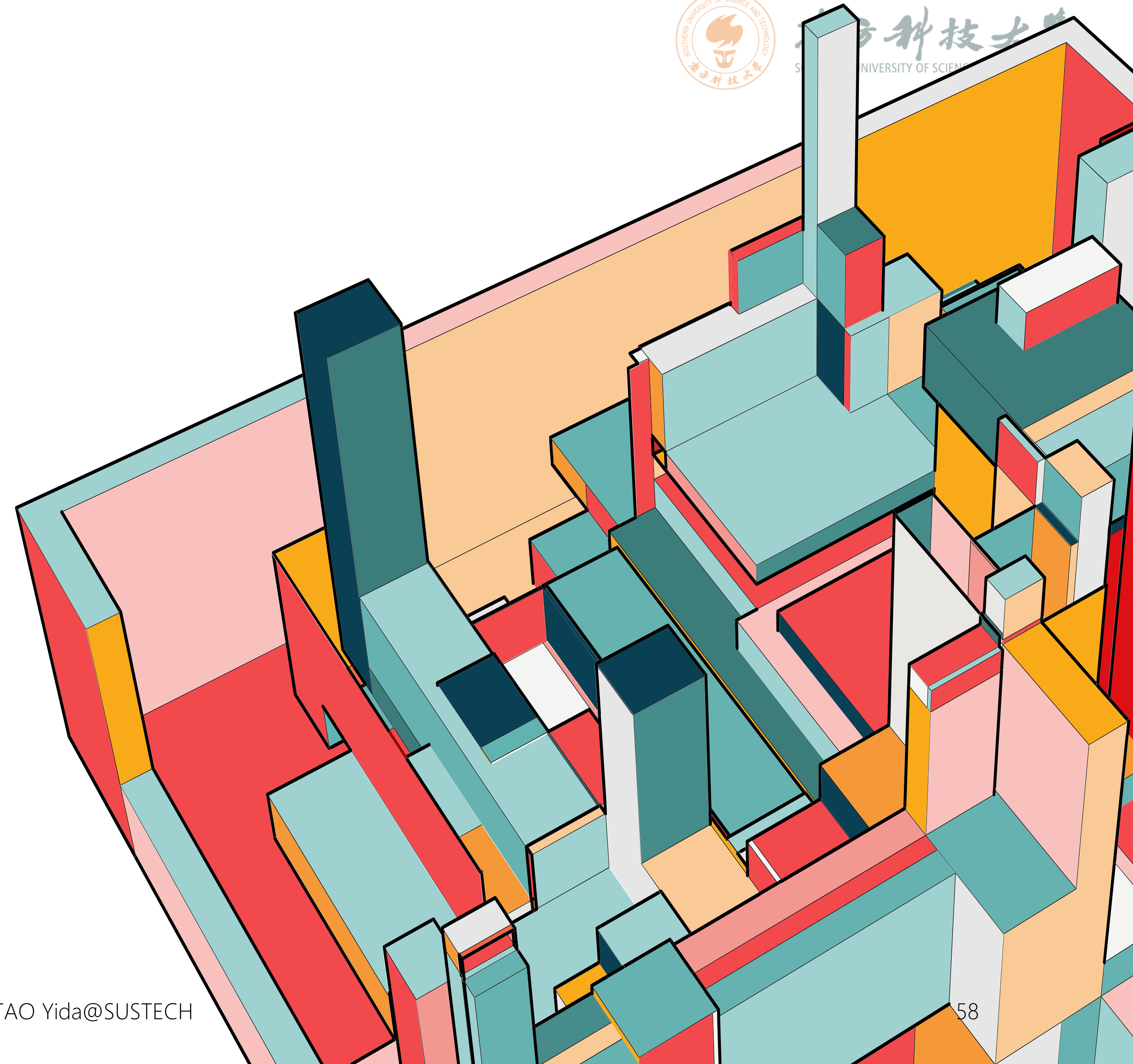
- A fuzzer can be
  - generation-based: inputs are generated from scratch
  - mutation-based: inputs are created by modifying existing inputs.
- A fuzzer can be whitebox or black-box, depending on whether it is aware of program structure.



https://blog.trailofbits.com/category/fuzzing/page/2/

READINGS

- Chapter 11-13. Software Engineering at Google by Winters et al
- Chapter 21. Software Testing: A Craftsman's Approach. Paul C. Jorgensen.
- 第9章 软件测试. 现代软件工程基础 by 彭鑫 et al.



NEXT

- Software Documentation