DAVE LIDDAMENT

TEST SUITE HOLY TRINITY

LET'S START WITH A STORY...

WHY ARE WE HERE

- What went wrong
- Why testing will help
- How can we build a good test suite
  - Only talking asserting correct functionality

Dave Liddament

@daveliddament

Lamp Bristol

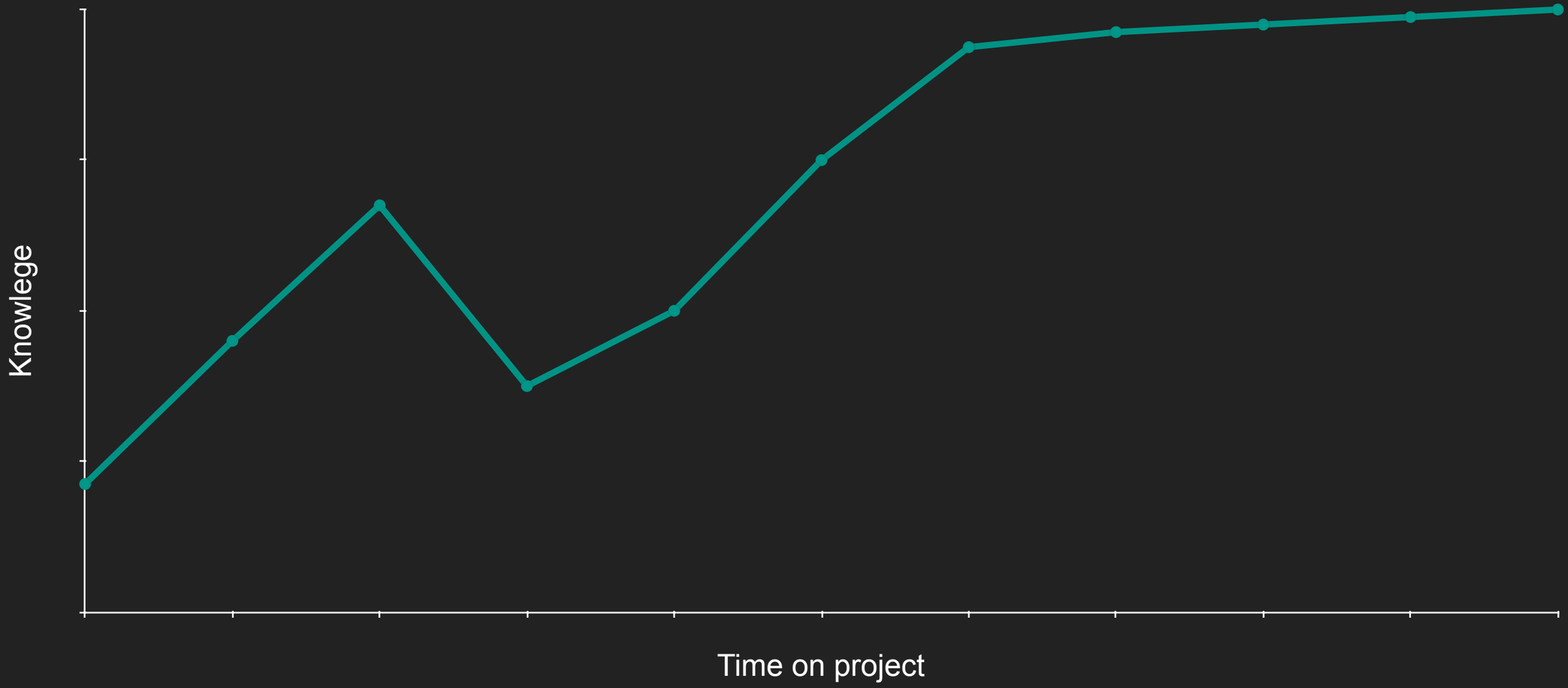
Organise PHP-SW and Bristol PHP Training

WHAT WENT WRONG?

TESTING FEEDBACK LOOP WAS TOO SLOW

REFACTORING WAS TOO RISKY

WE WILL ALL MAKE POOR DECISIONS AT THE START OF A PROJECT



WE NEED A TEST SUITE

- Prove that code works
- Prevent regression
- Allow us to refactor

IDEAL TEST SUITE

Fast

High coverage

Low maintenance

THE IDEAL TEST SUITE

- Fast
- High coverage
- Low maintenance

TERMINOLOGY

SYSTEM TEST

SYSTEM TEST

AWARD WINNING SOFTWARE

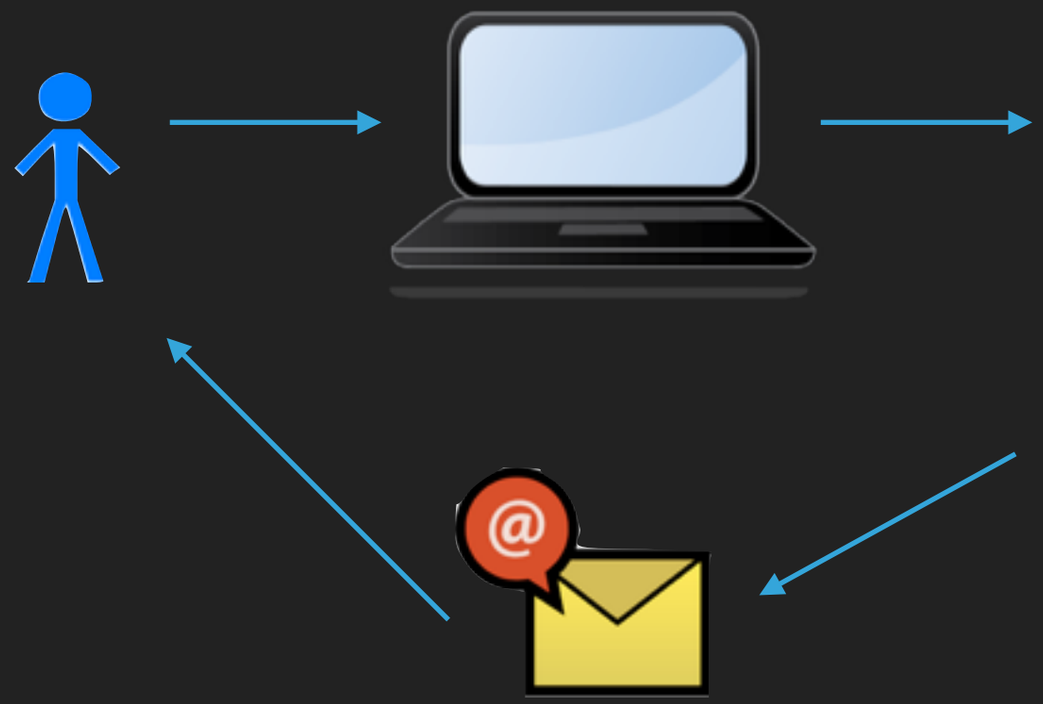
SYSTEM TEST

AWARD WINNING SOFTWARE

SYSTEM TEST

AWARD WINNING SOFTWARE

SYSTEM TEST



AWARD WINNING SOFTWARE

SYSTEM TEST

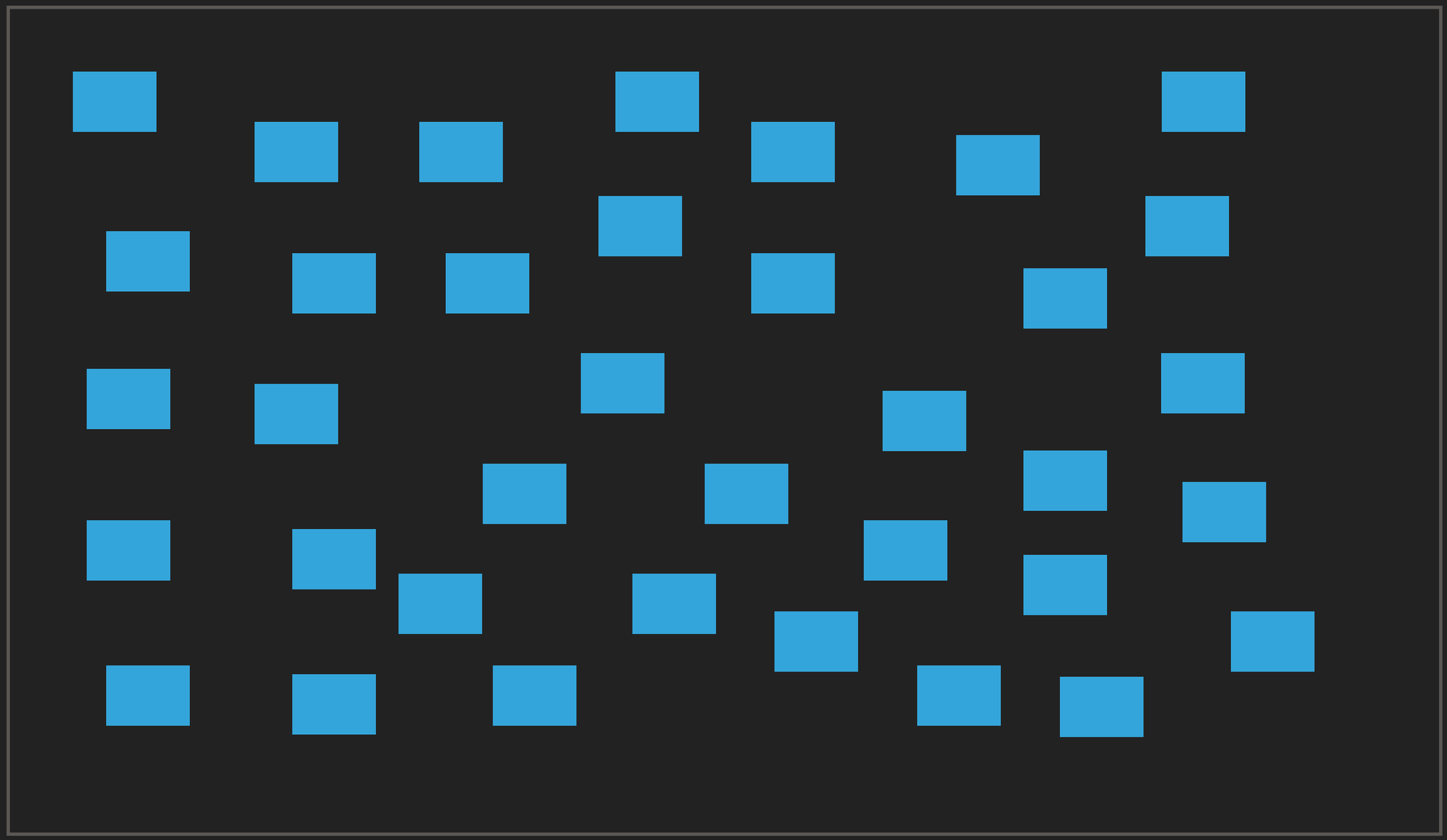
AWARD WINNING SOFTWARE

UNIT TEST

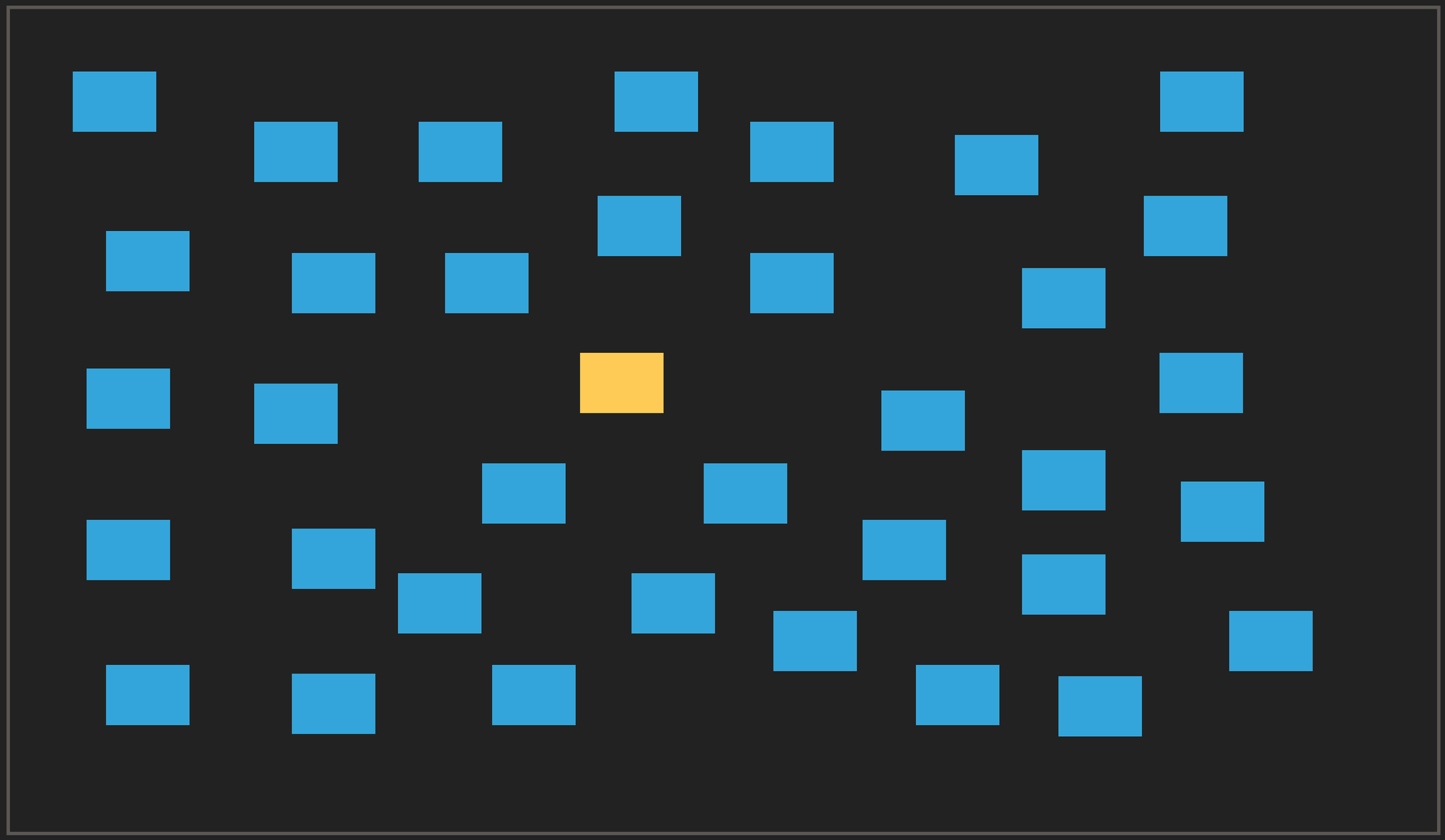
UNIT TEST

AWARD WINNING SOFTWARE

UNIT TEST



UNIT TEST



UNIT TEST EXAMPLE – SOFTWARE UNDER TEST

```
class PasswordValidator
{

   /**
    * Returns true if password meets following criteria:
    *
    * - 8 or more characters
    * - at least 1 digit
    * - at least 1 upper case letter
    * - at least 1 lower case letter
    */
    public function isValid(string $password) : bool
```

UNIT TEST EXAMPLE – TEST CASES REQUIRED

- Valid passwords:
  - “Passw0rd”
- Invalid passwords:
  - “Passw0r” - too short (everything else is good)
  - “Password” - no digit
  - “passw0rd” - no upper case letters
  - “PASSW0RD” - no lower case letters

THESE TESTS ARE FAMOUS FIVE

UNIT TEST EXAMPLE – TEST CASE (1)

```
class PasswordValidatorTest extends TestCase
{

    public function dataProvider() : array
    {
        return [
            [ “valid”       => [ true,  “Passw0rd” ],
            [ “tooShort”    => [ false, “Passw0r”  ],
            [ “noDigit”     => [ false, “Password” ],
            [ “noUpperCase” => [ false, “passw0rd” ],
            [ “noLowerCase” => [ false, “PASSW0RD” ],
        ];
    }

…
```

UNIT TEST EXAMPLE – TEST CASE (2)

...

```
    /**
     * @dataProvider dataProvider
     */
    public function testValidator(
        bool $expectedResult,
        string $inputValue
    ) {

        $validator = new PasswordValidator();
        $actualResult = $validator->isValid($inputValue);
        $this->assertEquals($expectedResult, $actualResult);
    }
```

TAKE AWAY

USE DATA PROVIDERS FOR UNIT TESTS

- Unit test sweet spot
- Quicker to test than not test

TAKE AWAY

USE DATA PROVIDERS FOR UNIT TESTS

- Unit test sweet spot
- Quicker to test than not test

```
class PasswordValidatorTest extends TestCase
{
   public function dataProvider() : array
   {
      return [
        [ “valid”       => [ true,  “Passw0rd” ],
        [ “tooShort”    => [ false, “Passw0r”  ],
        [ “noDigit”     => [ false, “Password” ],
        [ “noUpperCase” => [ false, “passw0rd” ],
        [ “noLowerCase” => [ false, “PASSW0RD” ],
       ];
  }

   /**
    * @dataProvider dataProvider
    */
   public function testValidator(bool $expectedResult, string $inputValue) {
          $validator = new PasswordValidator();
          $actualResult = $validator->isValid($inputValue);
          $this->assertEquals($expectedResult, $actualResult);
    }
}
```

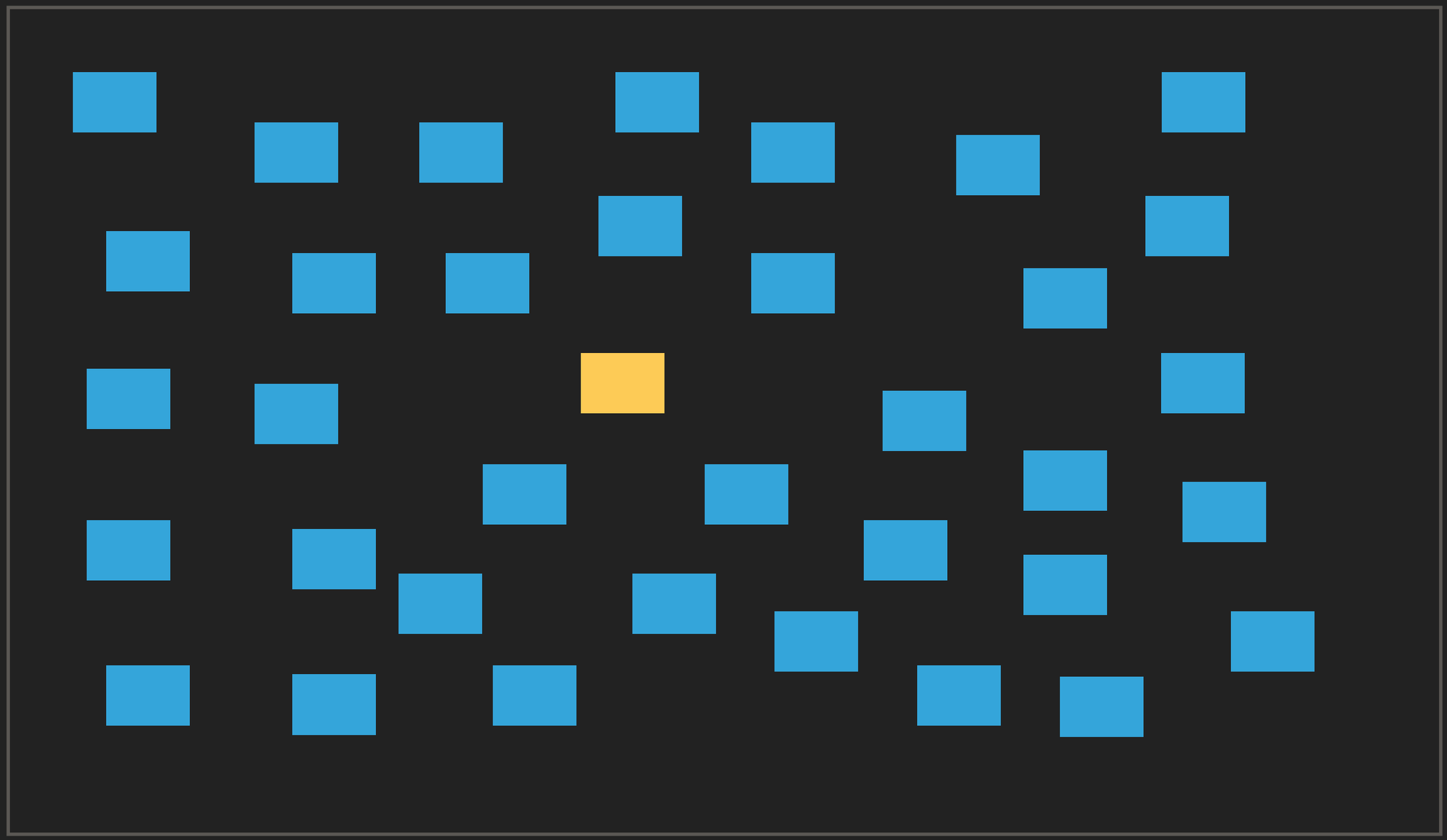
TESTING CONTINUUM

TESTING CONTINUUM

Unit tests

Systems tests

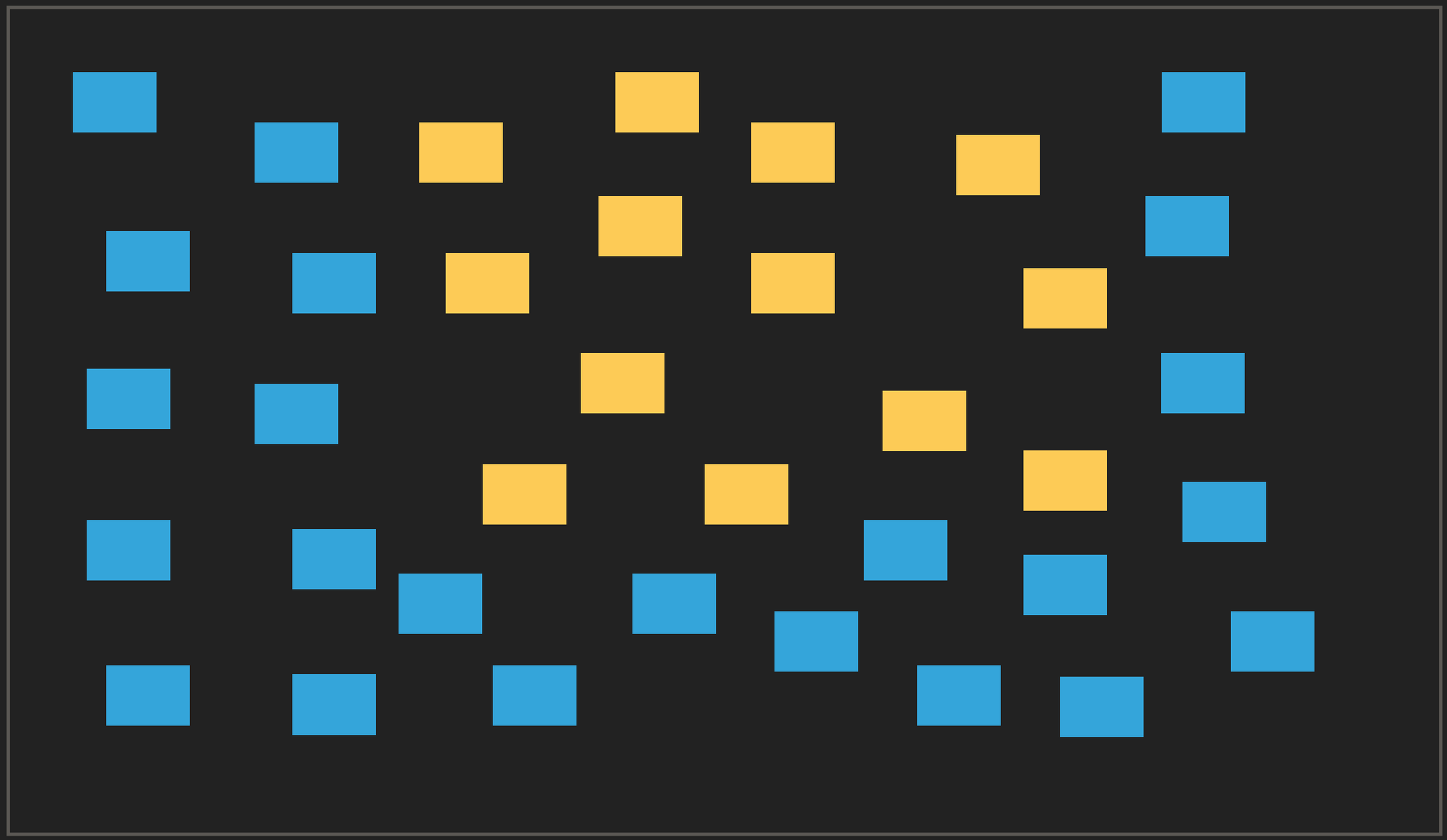
TESTING CONTINUUM



Unit tests

Systems tests

TESTING CONTINUUM



Unit tests

Systems tests

TESTING CONTINUUM

AWARD WINNING SOFTWARE

Unit tests

Systems tests

THE IDEAL TEST SUITE

- Fast
- High coverage
- Low maintenance

SPEED OF EXECUTION

Unit tests

Systems tests

SPEED OF EXECUTION

Fast

Unit tests

Systems tests

SYSTEM TEST

AWARD WINNING SOFTWARE

SPEED OF EXECUTION

Fast

Slow

Unit tests

Systems tests

COVERAGE

Unit tests

Systems tests

COVERAGE

High

Unit tests

Systems tests

COVERAGE

High

Low

Unit tests

Systems tests

COVERAGE

Low

High

Low

Unit tests

Systems tests

COVERAGE

Low

High

High

Low

Unit tests

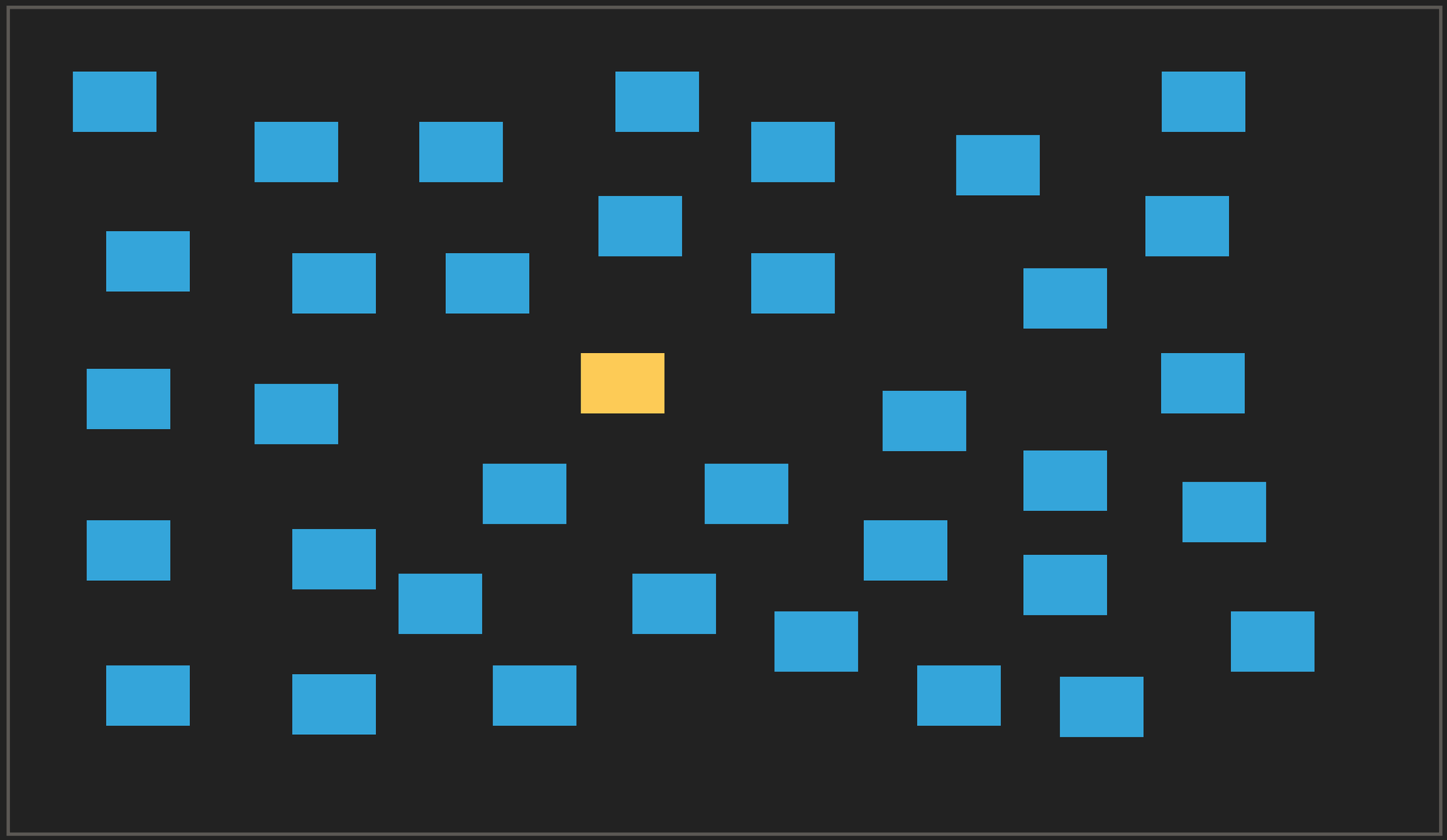
Systems tests

MAINTENANCE COSTS

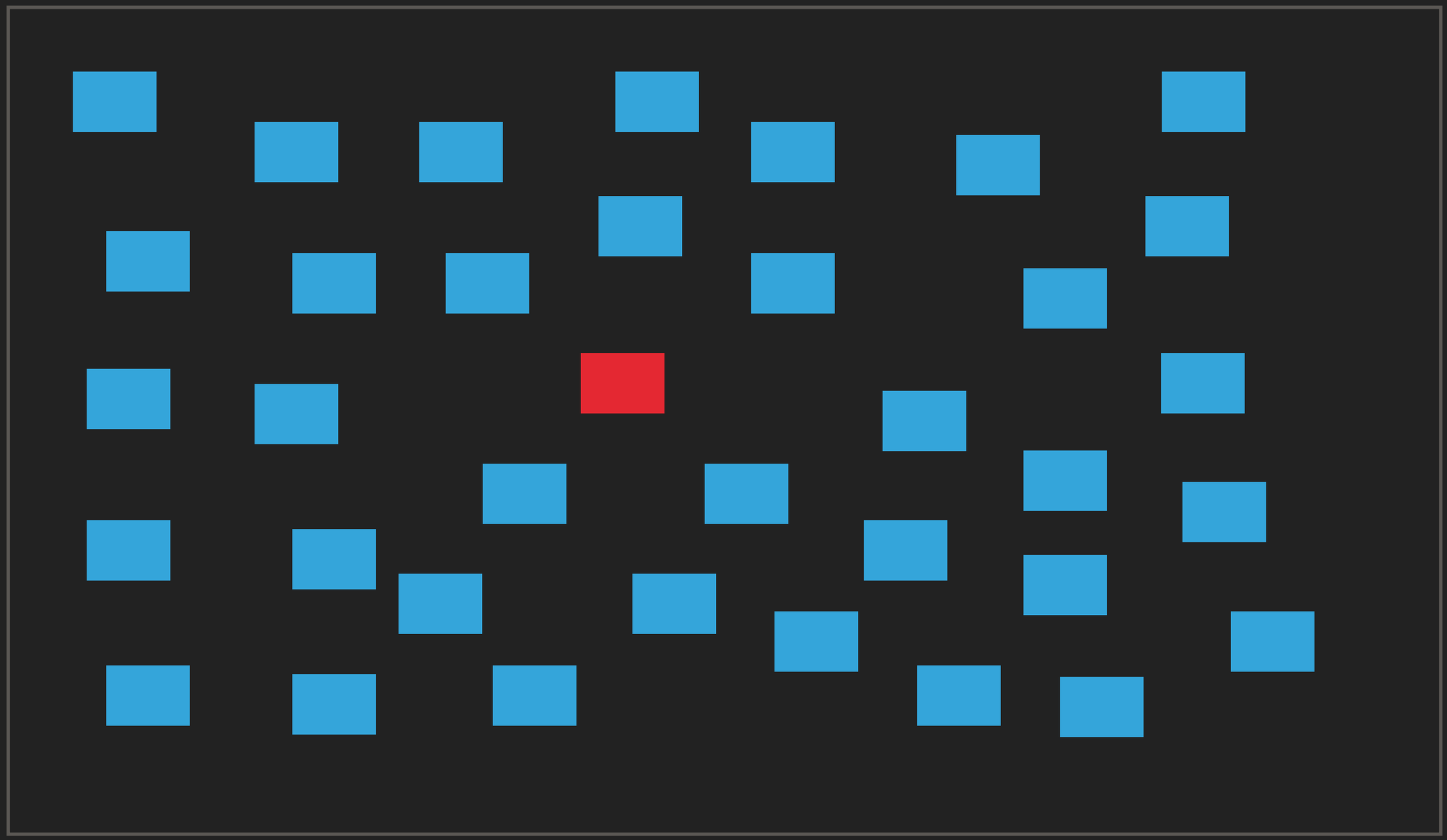
Unit tests

Systems tests

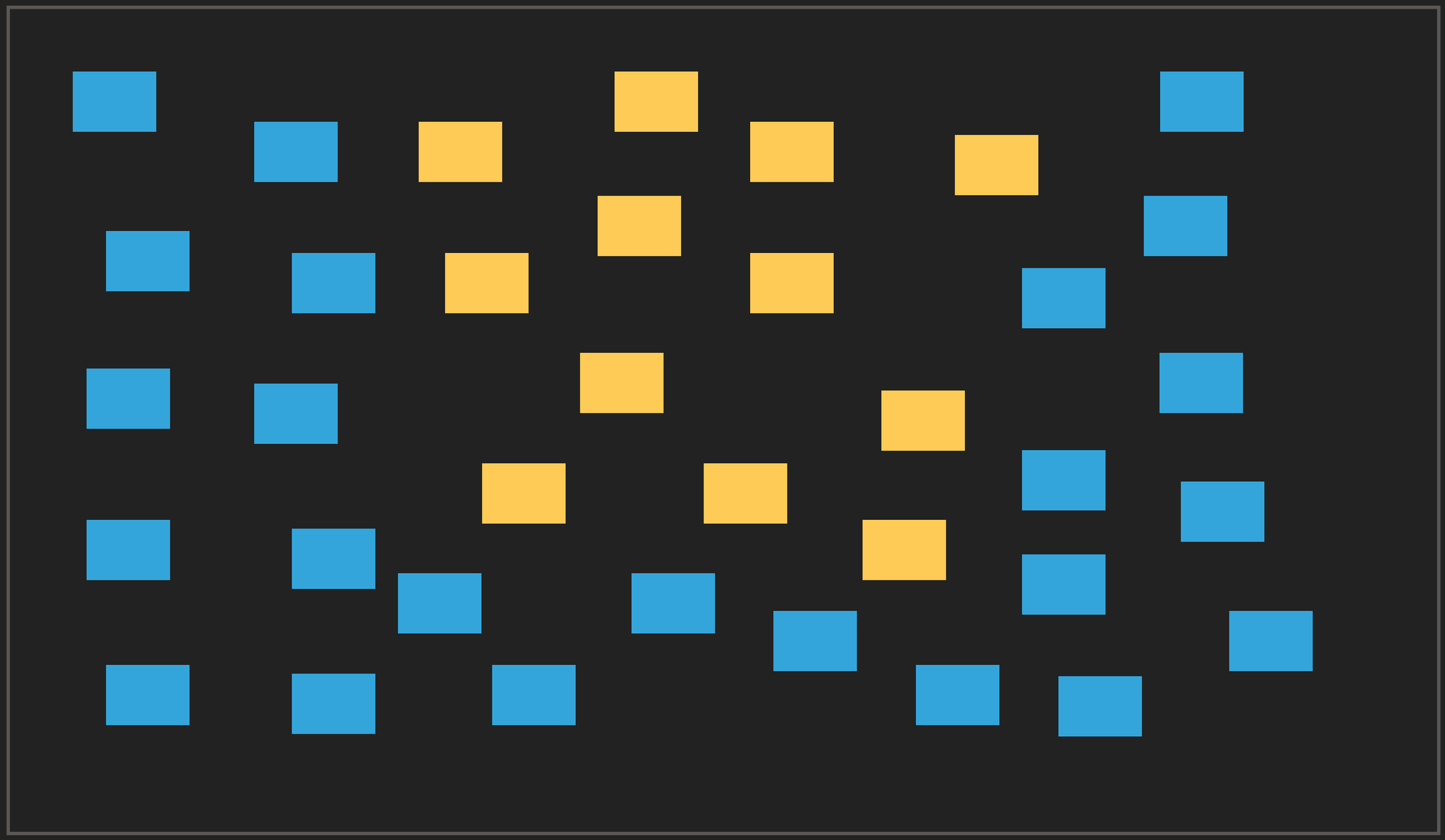
UNIT TEST



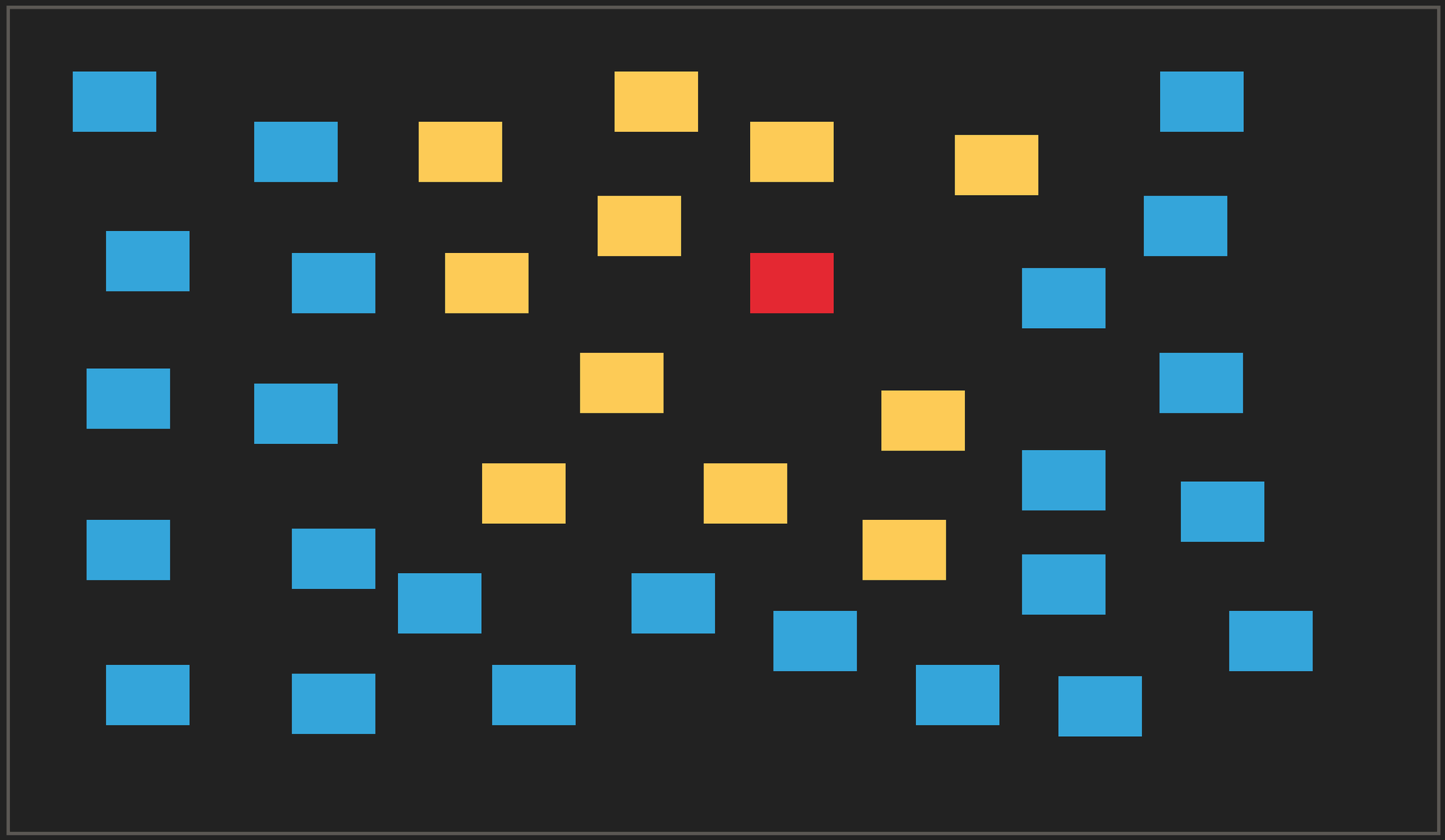
UNIT TEST



INTEGRATION TEST



INTEGRATION TEST



MAINTENANCE COSTS

Unit tests

Systems tests

MAINTENANCE COSTS

Low

Unit tests

Systems tests

MAINTENANCE COSTS

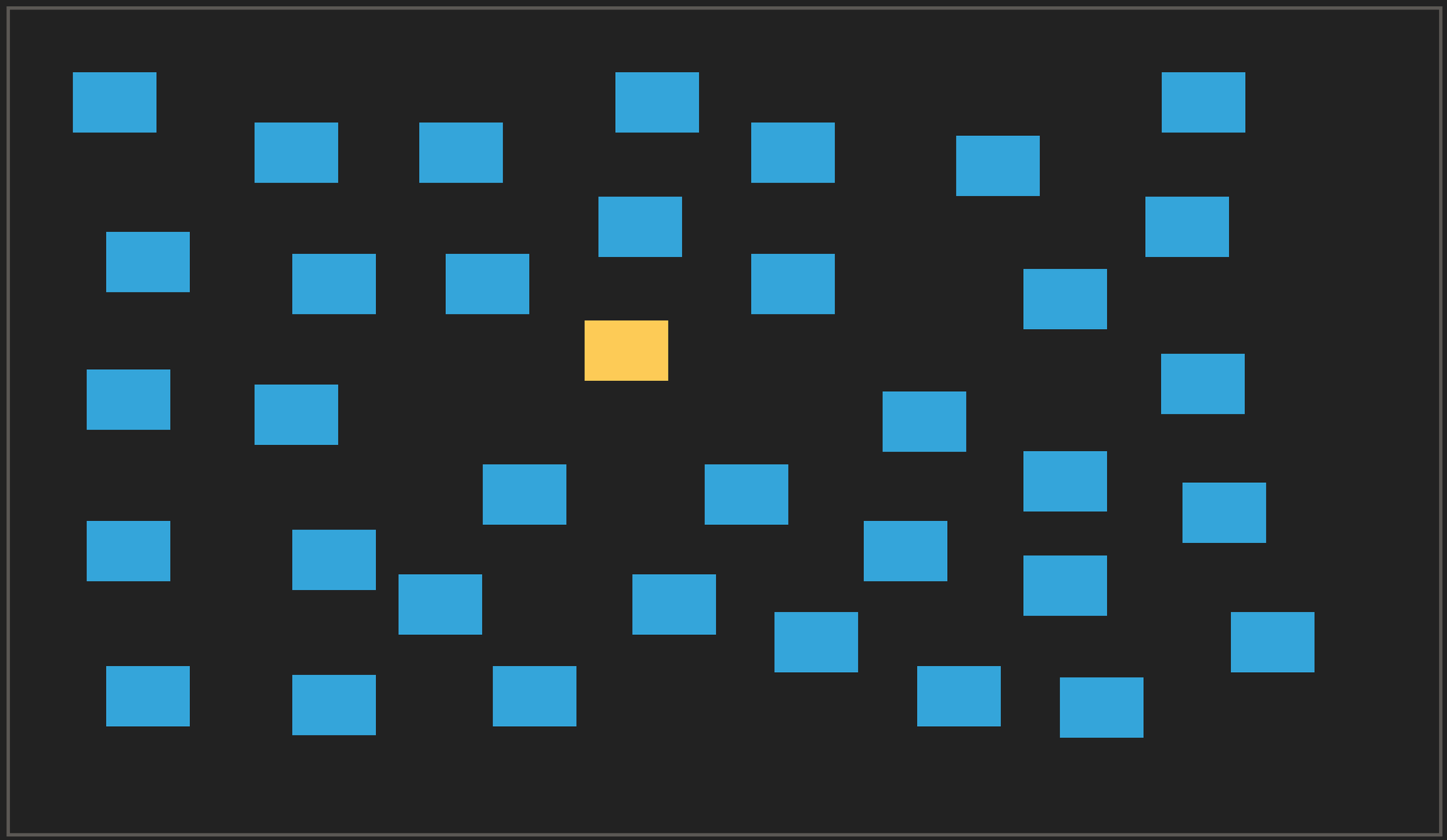
Low

High

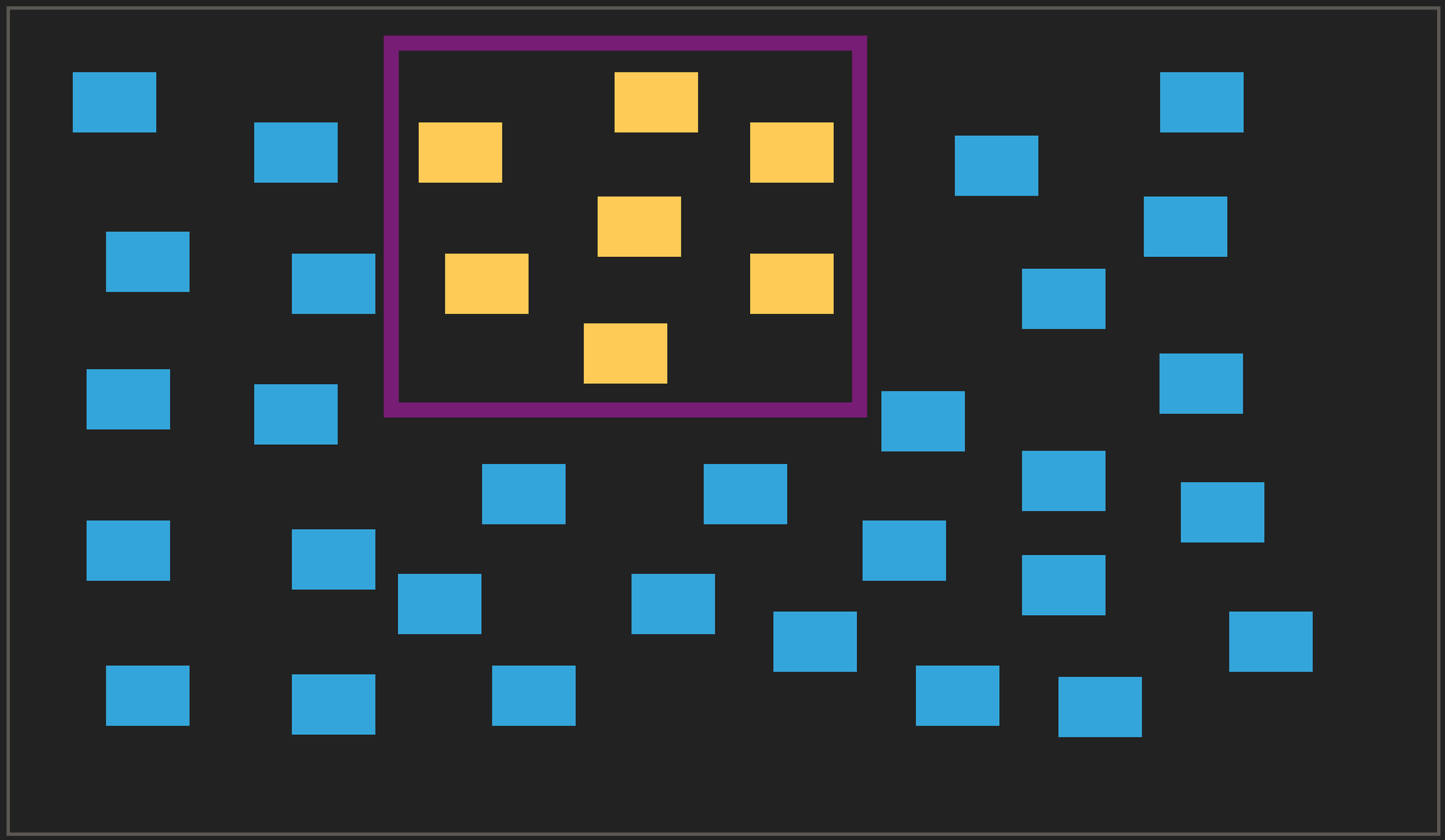
Unit tests

Systems tests

REFACTORING SCOPE



REFACTORING SCOPE



MAINTENANCE COSTS

Unit tests

Systems tests

MAINTENANCE COSTS

High

Unit tests

Systems tests

MAINTENANCE COSTS

High

Low

Unit tests

Systems tests

THANK GOODNESS FOR THAT MOMENTS

Unit tests

Systems tests

THANK GOODNESS FOR THAT MOMENTS

Low

Unit tests

Systems tests

THANK GOODNESS FOR THAT MOMENTS

Low

High

Unit tests

Systems tests

SO FAR NOTHING TOO CONTROVERSIAL

NOTHING IS BLACK AND WHITE

EVERYTHING IS COMPROMISE

WRITING A GOOD TEST SUITE IS A SKILL

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

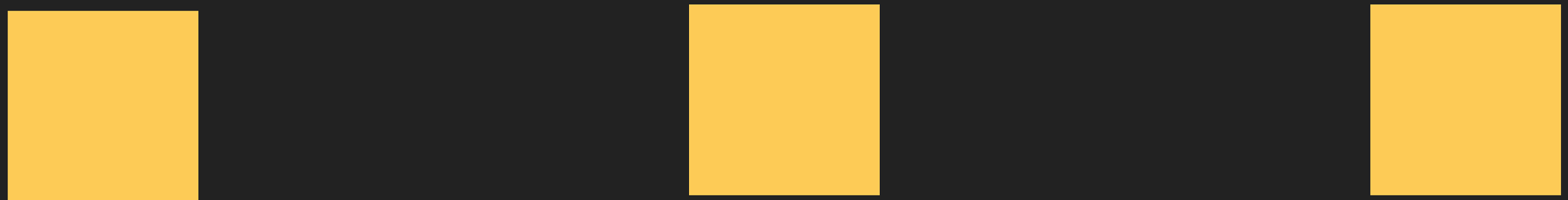
Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?



Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

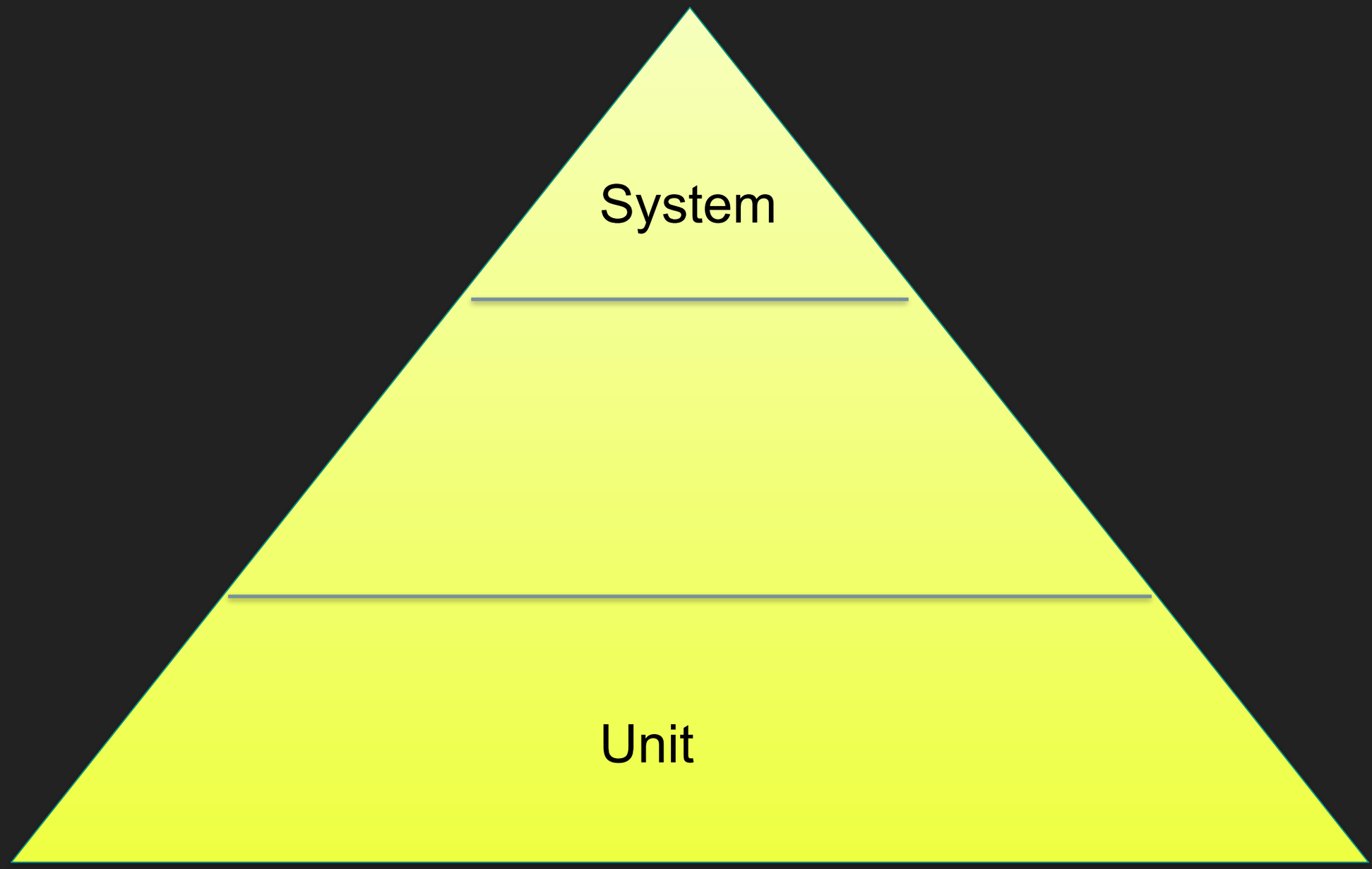
Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

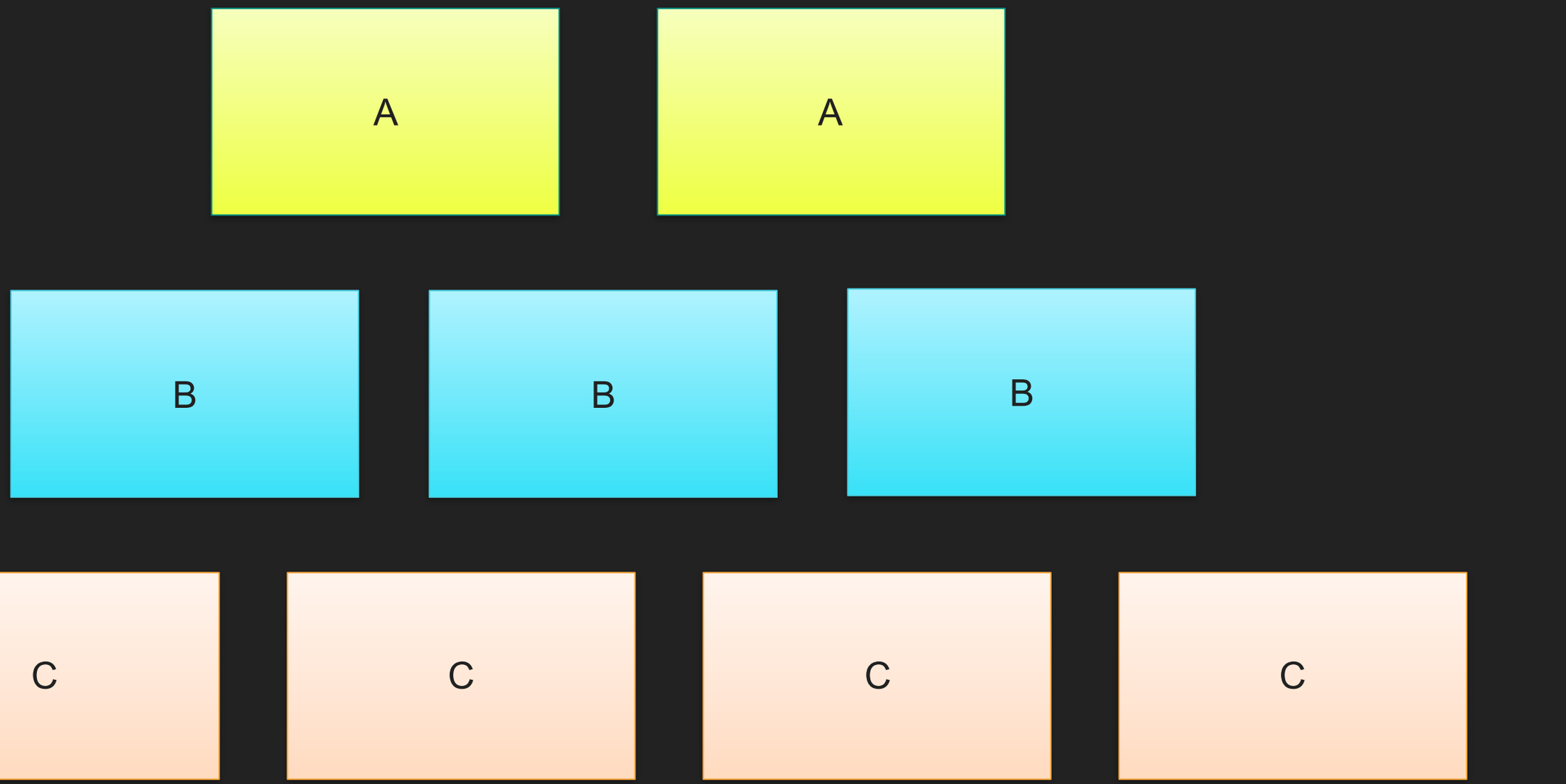
Unit tests

Systems tests

TEST PYRAMID

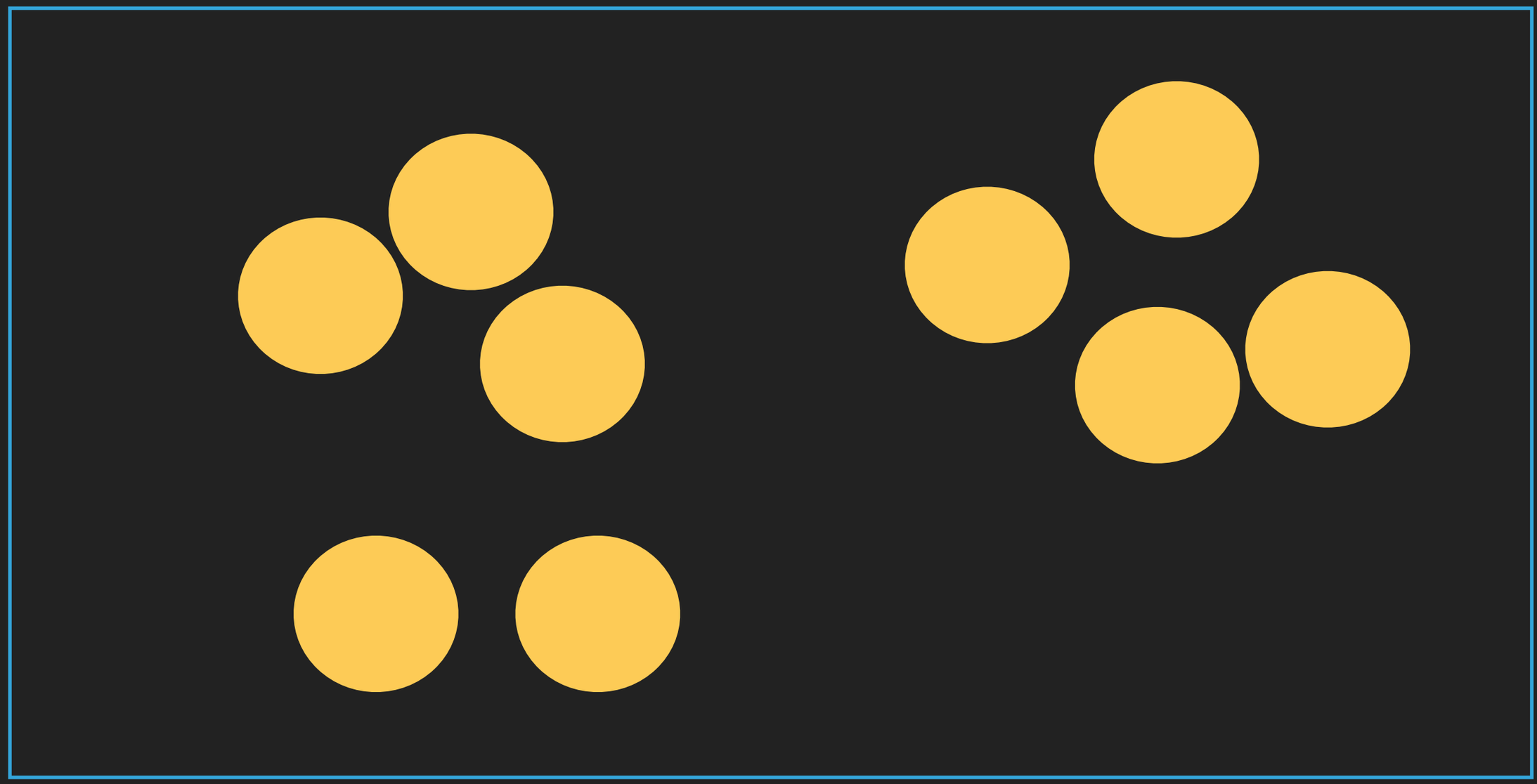


TEST IN LAYERS

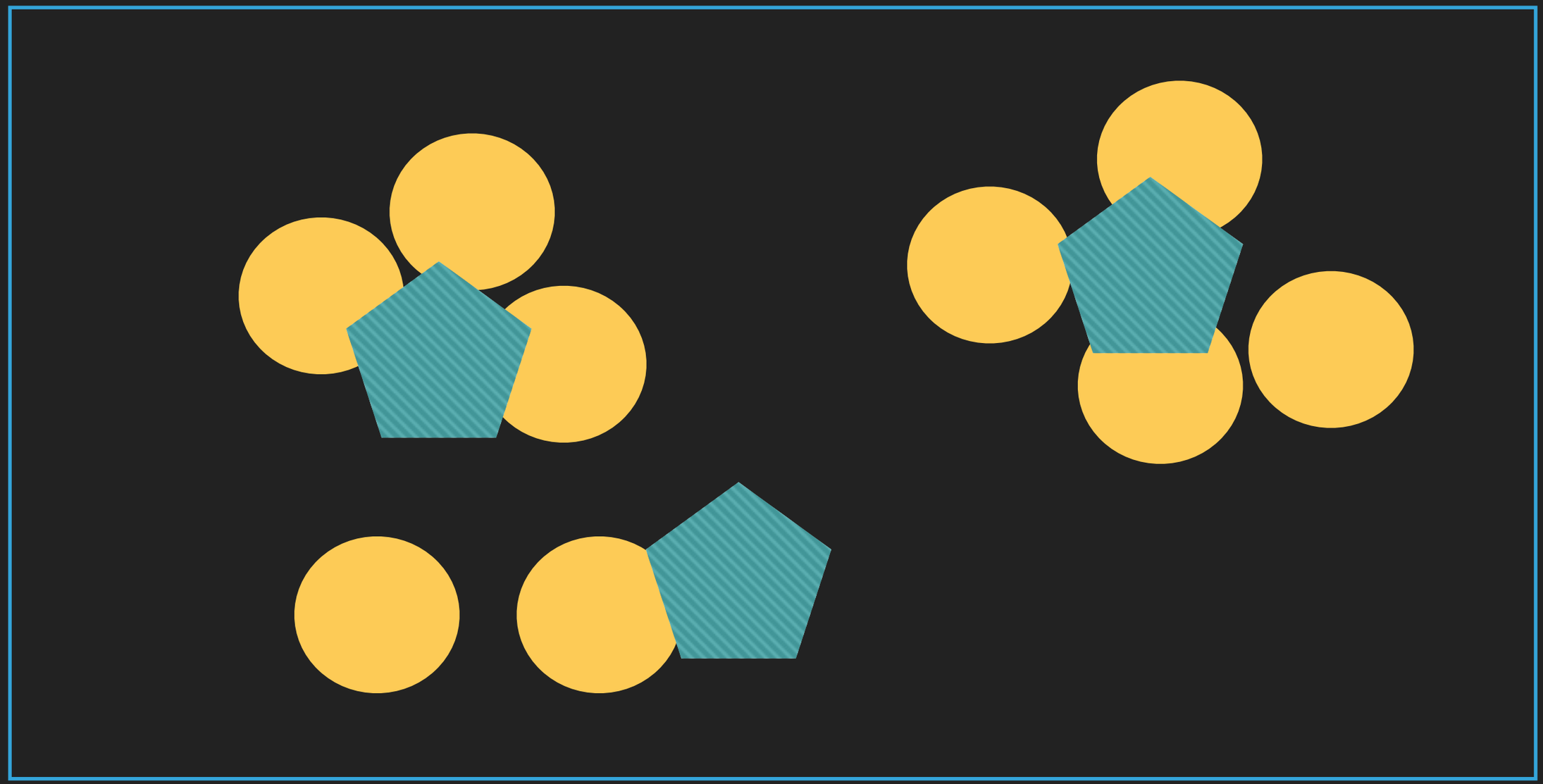


COVERAGE: SUM IS GREATER THAN THE PARTS*

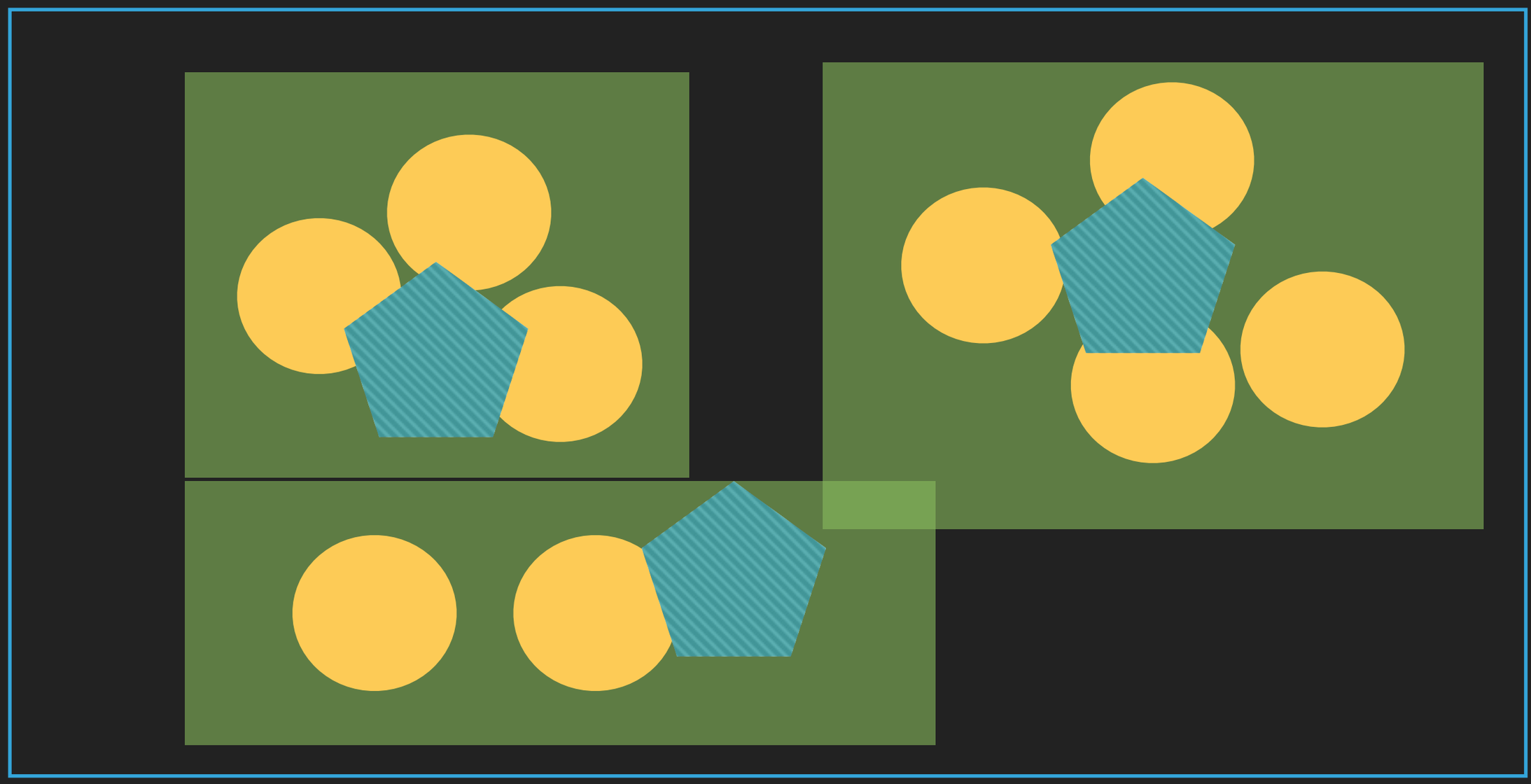
COVERAGE: SUM IS GREATER THAN THE PARTS*



COVERAGE: SUM IS GREATER THAN THE PARTS*



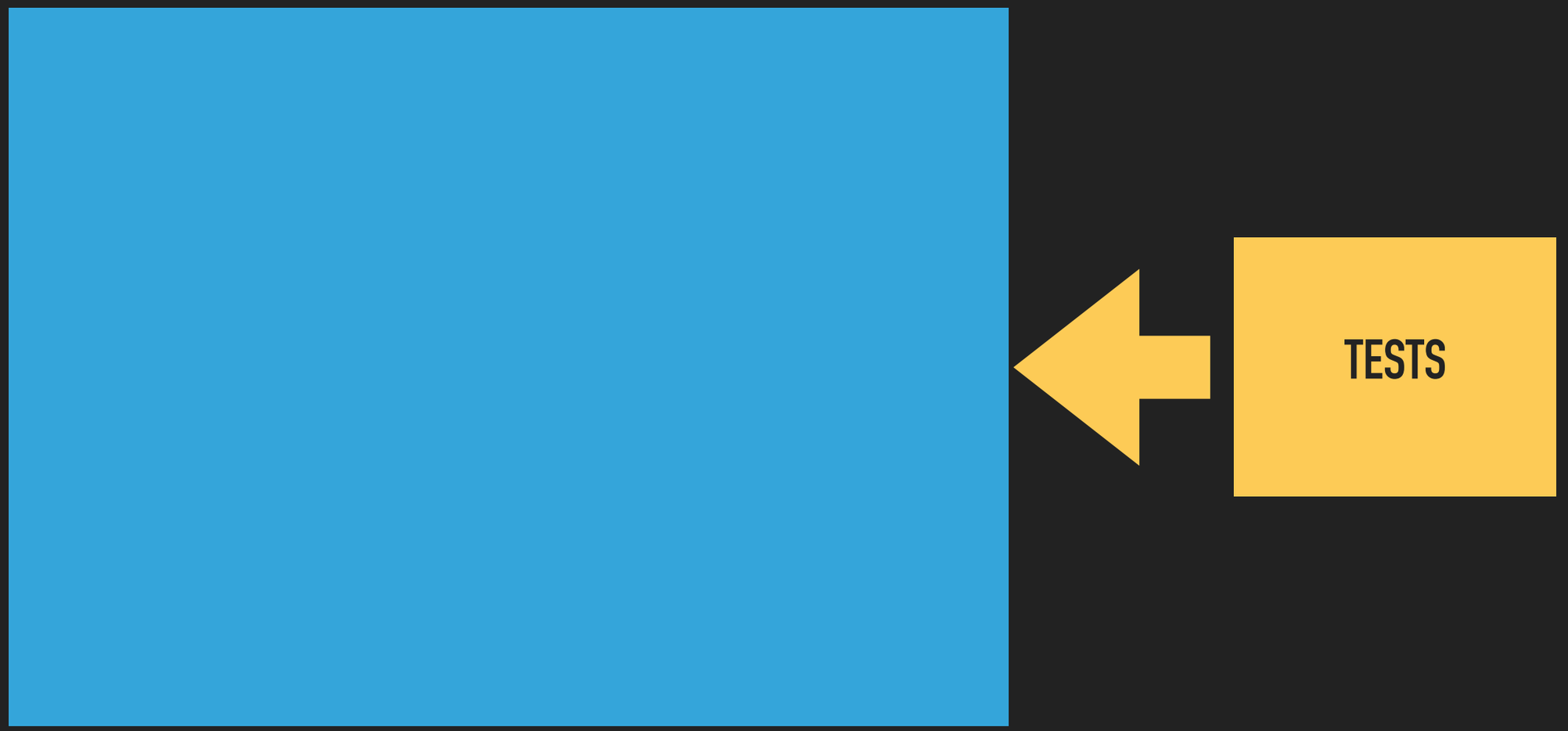
COVERAGE: SUM IS GREATER THAN THE PARTS*



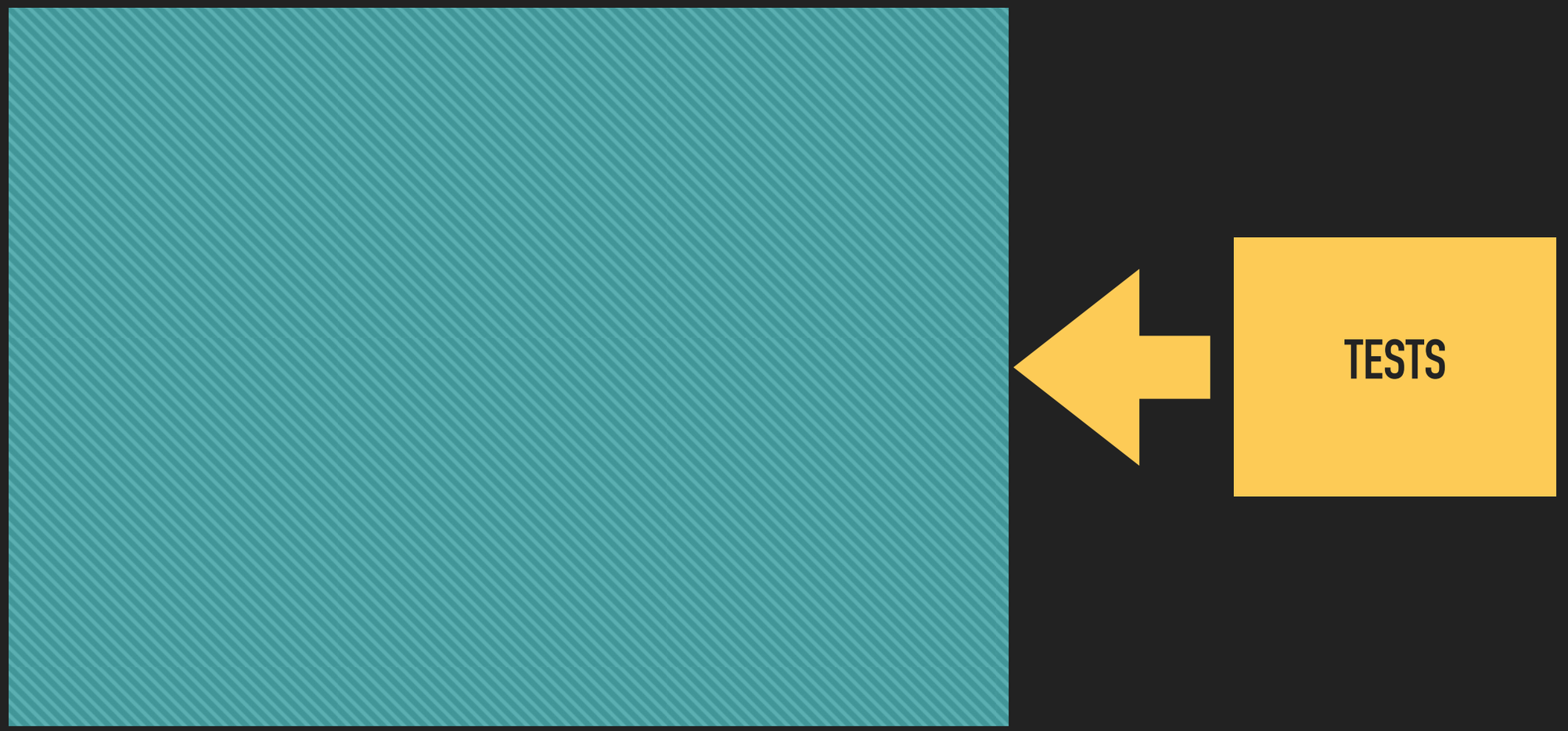
* WRITING A GOOD TEST SUITE IS A SKILL

TEST PYRAMID IS STILL A COMPROMISE

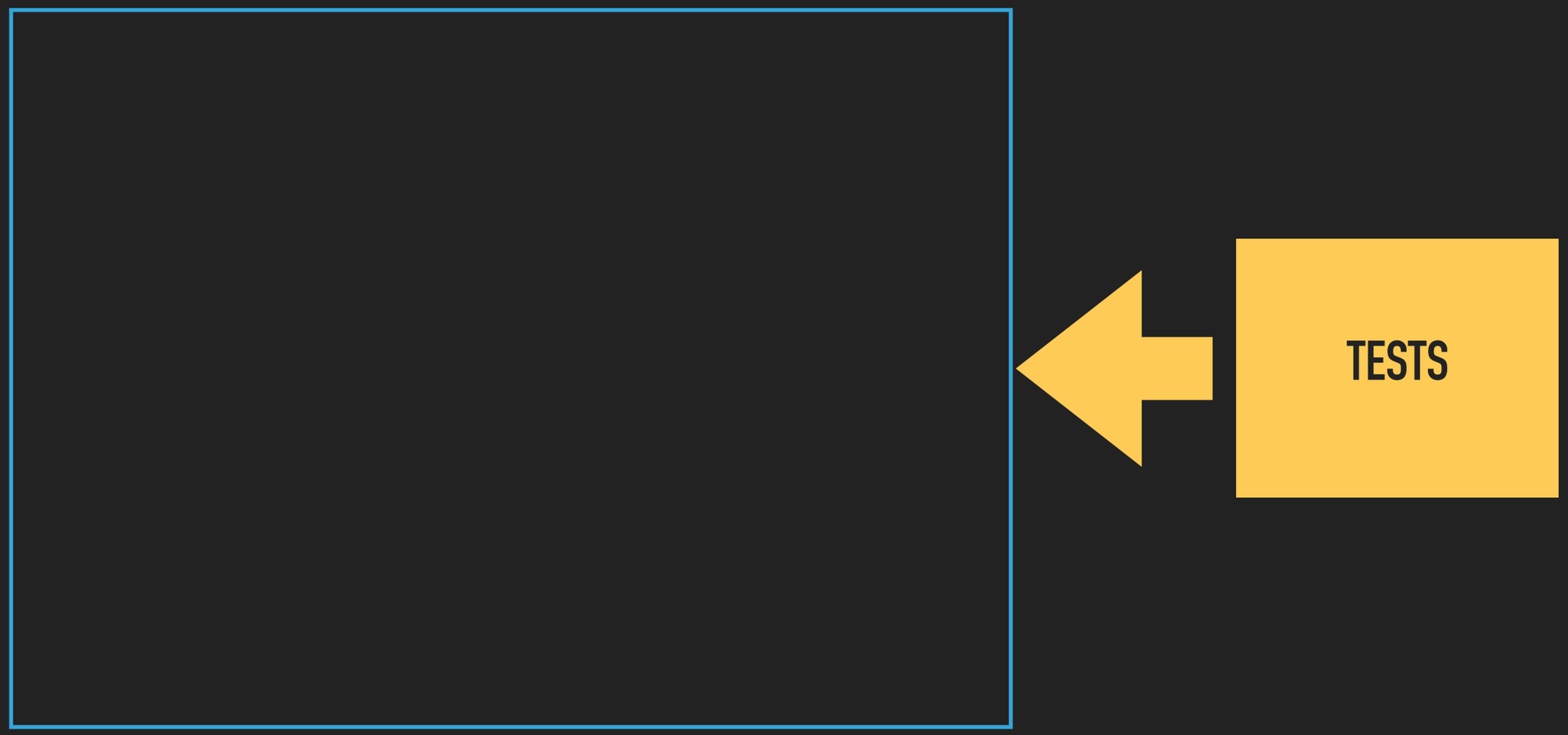
TEST PYRAMID IS STILL A COMPROMISE



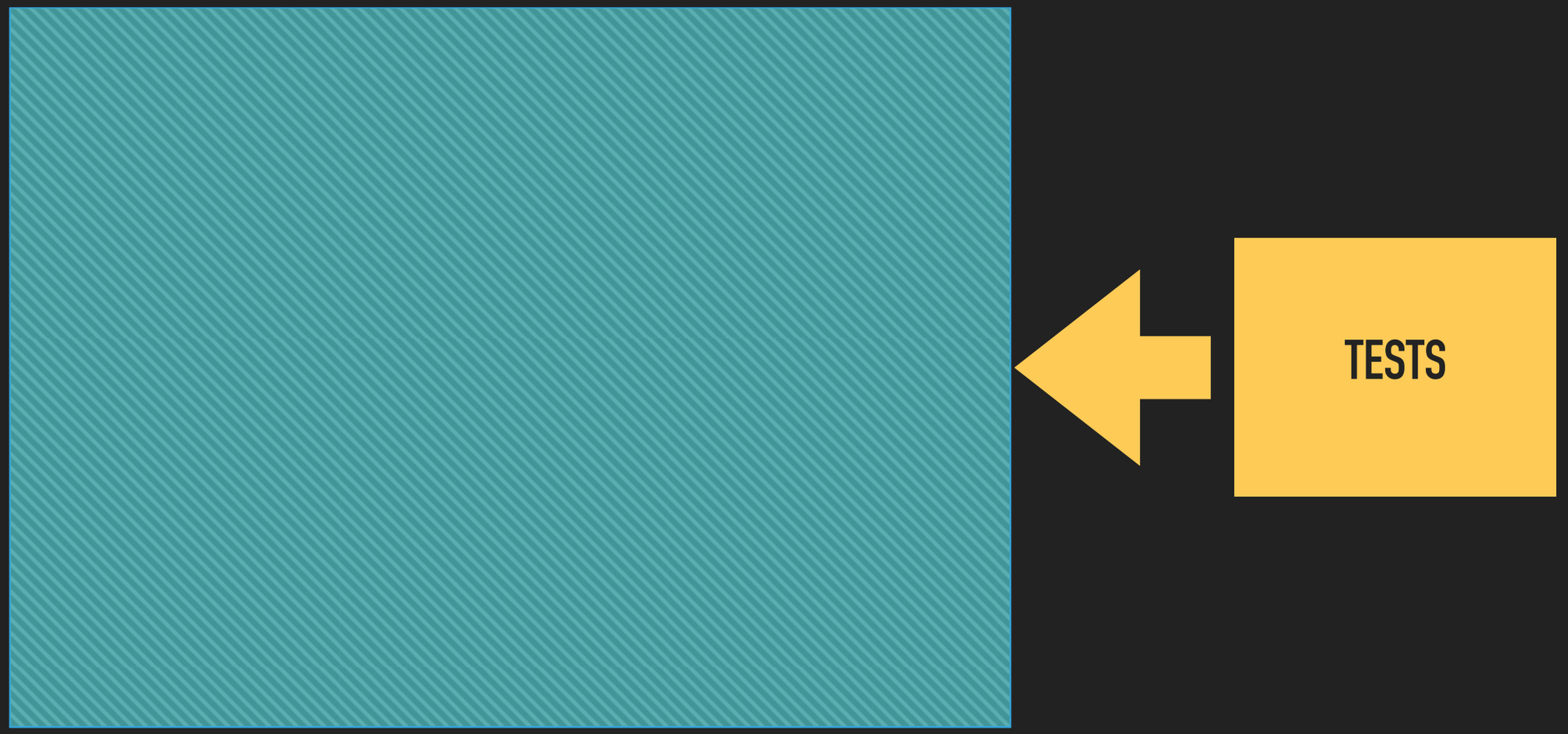
TEST PYRAMID IS STILL A COMPROMISE



TEST PYRAMID IS STILL A COMPROMISE



TEST PYRAMID IS STILL A COMPROMISE



WHY DO WE NEED A TEST SUITE

- Prove code works
- Prevent against regression
- Allow safe refactoring of code

OUR IDEAL TEST SUITE WOULD BE...

- Fast to execute
- High coverage
- Low maintenance

EVERY THING IS A COMPROMISE

- Nothing is black and white

UNIT TESTS IN MORE DEPTH

NEW REQUIREMENT

```
class PasswordValidator
{

   /**
    * Returns true if password meets following criteria:
    *
    * - 8 or more characters
    * - at least 1 digit
    * - at least 1 upper case letter
    * - at least 1 lower case letter
    * - not one of the user's previous 5 passwords
    */
    public function isValid(string $password, User $user) : bool
```

ARCHITECTURE

EXISTING PASSWORD VALIDATION RULES

ARCHITECTURE

EXISTING PASSWORD VALIDATION RULES

CHECK IF LAST 5 PASSWORDS

ARCHITECTURE

EXISTING PASSWORD VALIDATION RULES

CHECK IF LAST 5 PASSWORDS

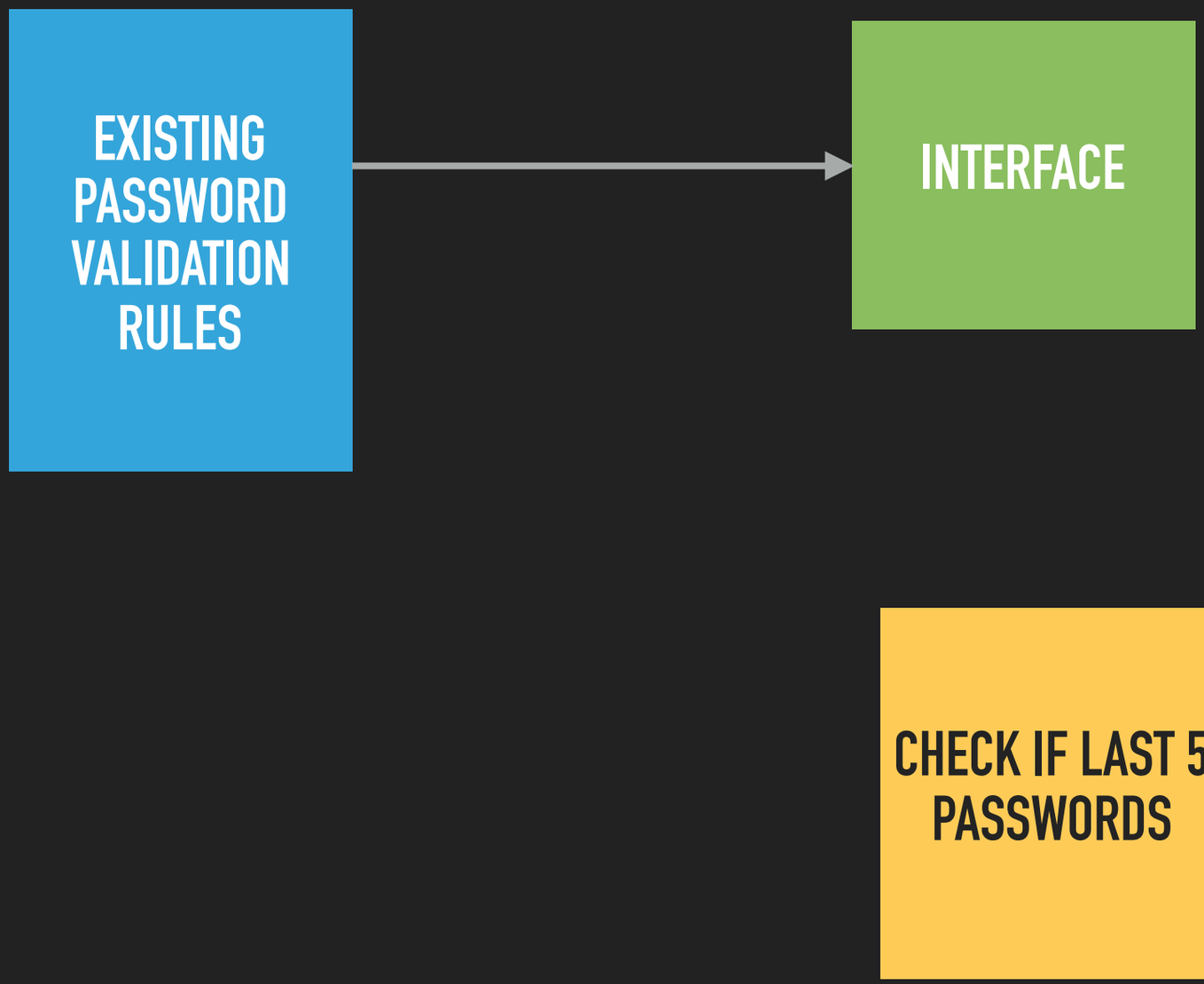
ARCHITECTURE

EXISTING PASSWORD VALIDATION RULES

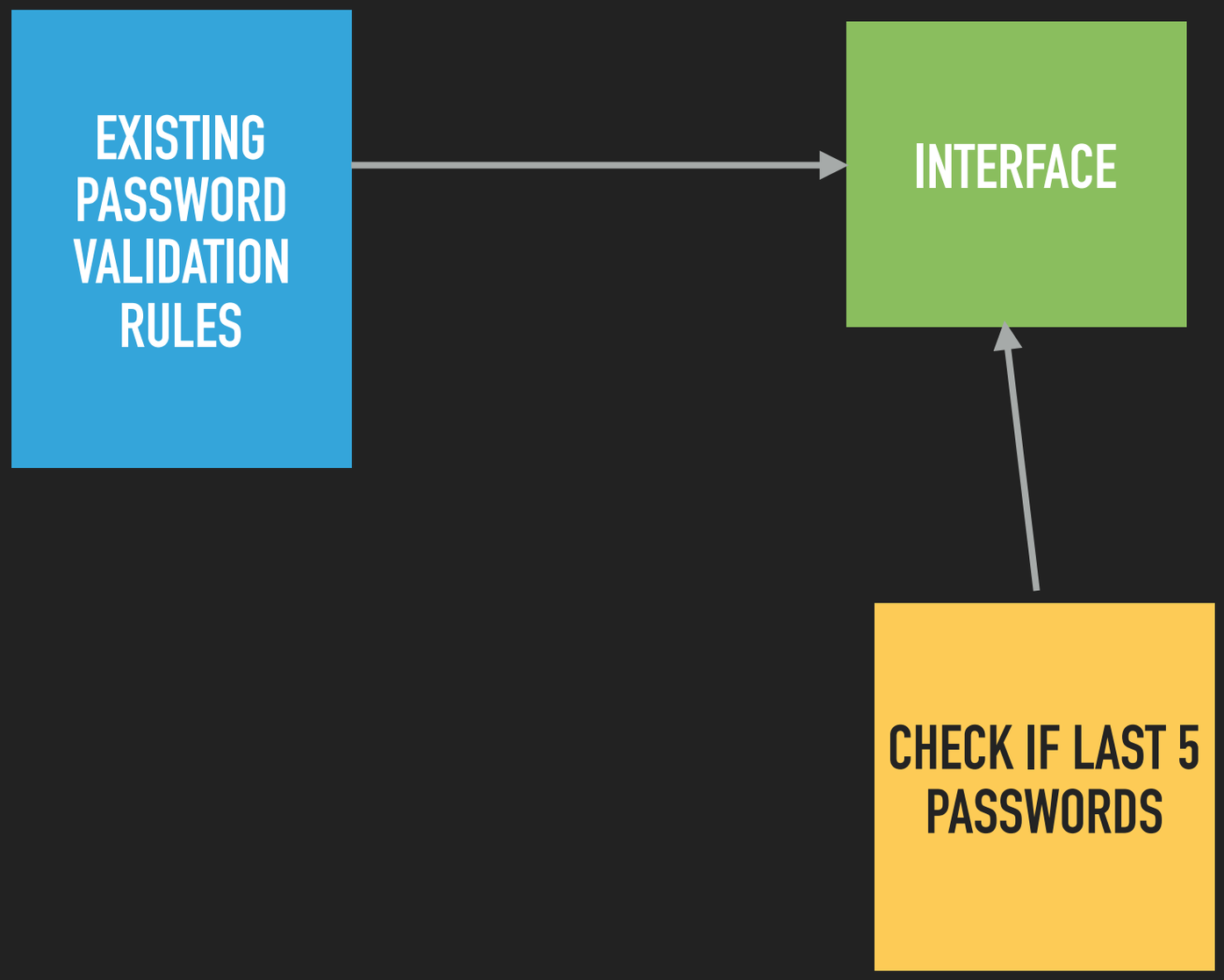
INTERFACE

CHECK IF LAST 5 PASSWORDS

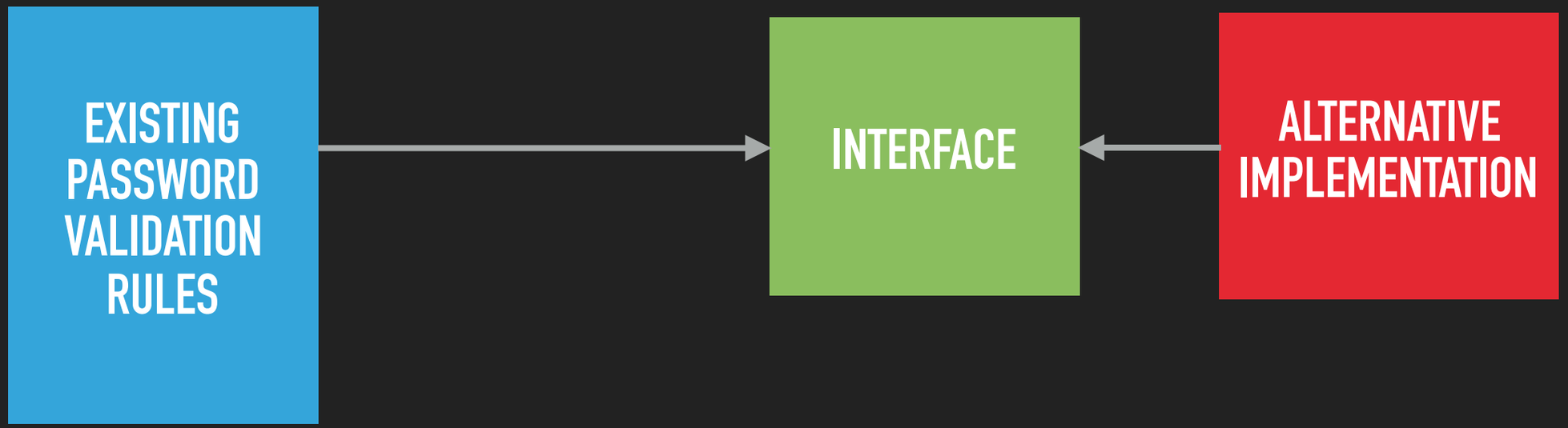
ARCHITECTURE



ARCHITECTURE



ARCHITECTURE



PREVIOUS PASSWORD CHECKER INTERFACE

```
interface PreviousPasswordChecker
{

    /**
     * Returns true if password has been used by user
     * in previous 5 passwords
     */
    public function isPreviouslyUsed($password, $user);

}
```

USE DEPENDENCY INJECTION

```
class PasswordValidator
{

  private $previousPasswordChecker

  public function __construct($previousPasswordChecker) {
    $this->previousPasswordChecker = $previousPasswordChecker;
  }

  public function isValid(string $password) : bool
  {
    …
  }
```

OPTIONS WITH DEPENDENCIES

- Real thing
- Test double
  - Stub
  - Mock
  - Fake

PASSWORD VALIDATOR TEST REVISITED

- Update existing tests to account for:
  - Updated PasswordValidator constructor
  - Any calls to RecentPasswordChecker
- New tests
  - Valid password. Has been recently used
  - Valid password. Has NOT been recently used

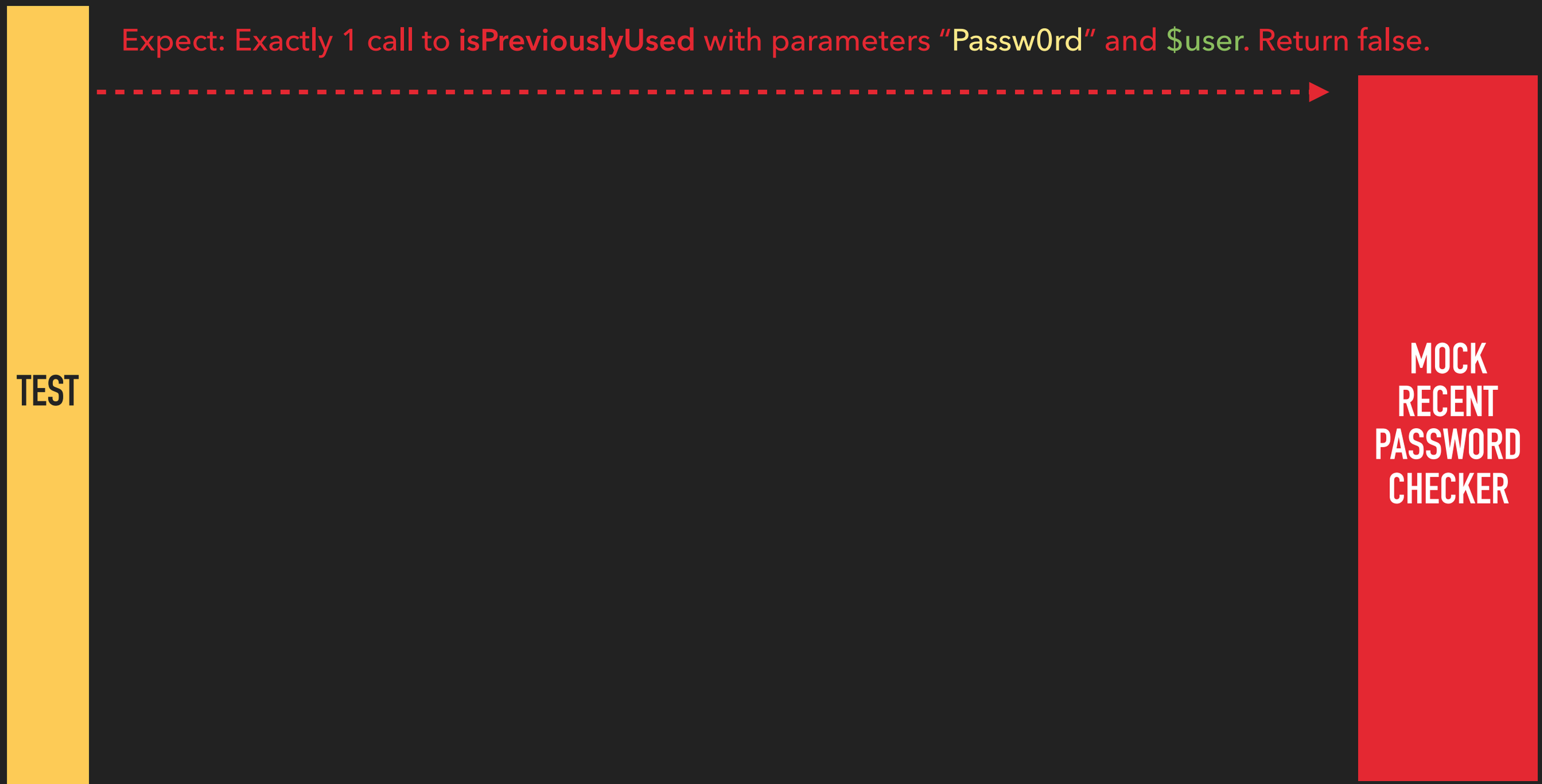
NEW TEST: VALID PASSWORD, NOT RECENTLY USED

NEW TEST: VALID PASSWORD, NOT RECENTLY USED

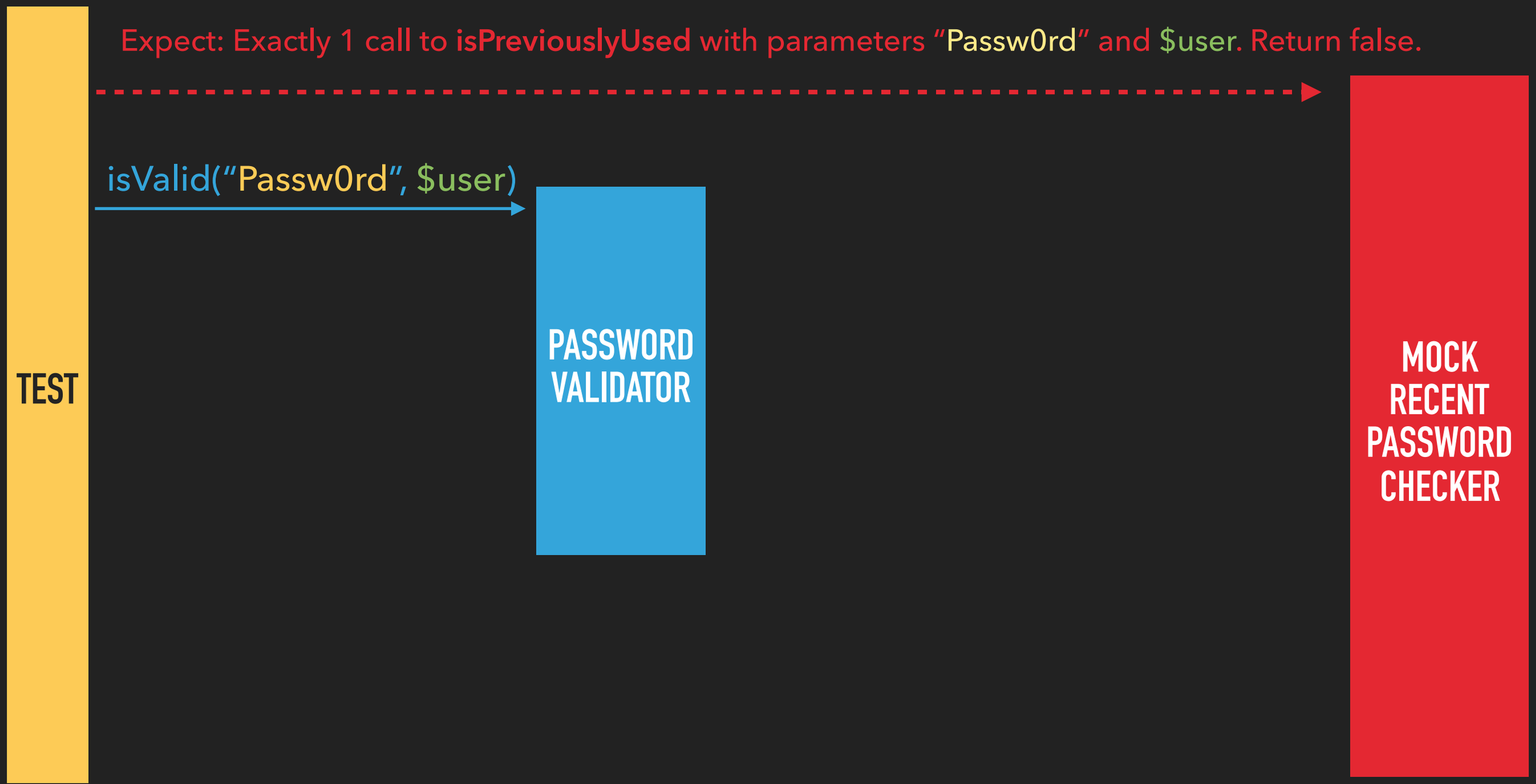
TEST

NEW TEST: VALID PASSWORD, NOT RECENTLY USED

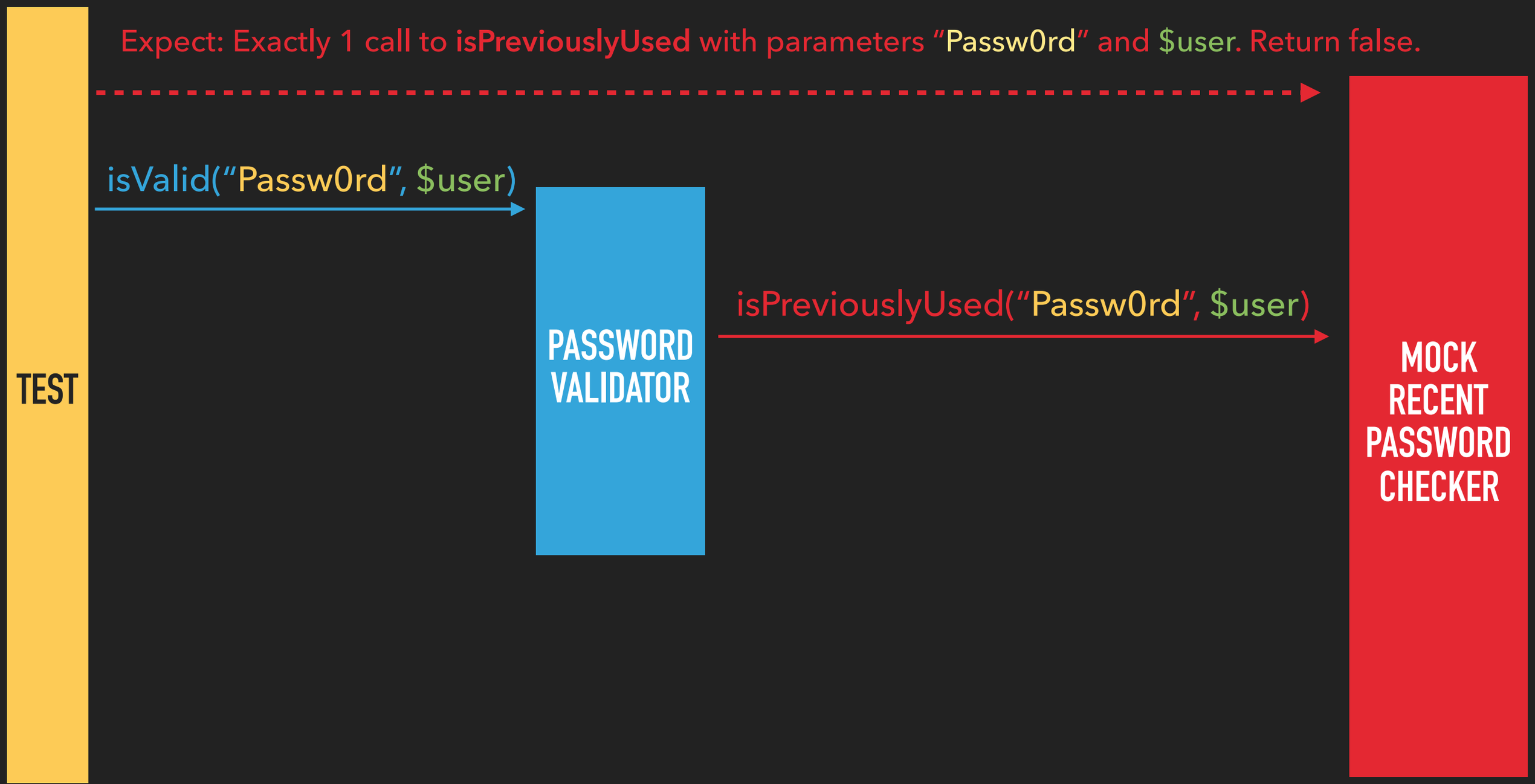
Expect: Exactly 1 call to **isPreviouslyUsed** with parameters "Passw0rd" and $user. Return false.



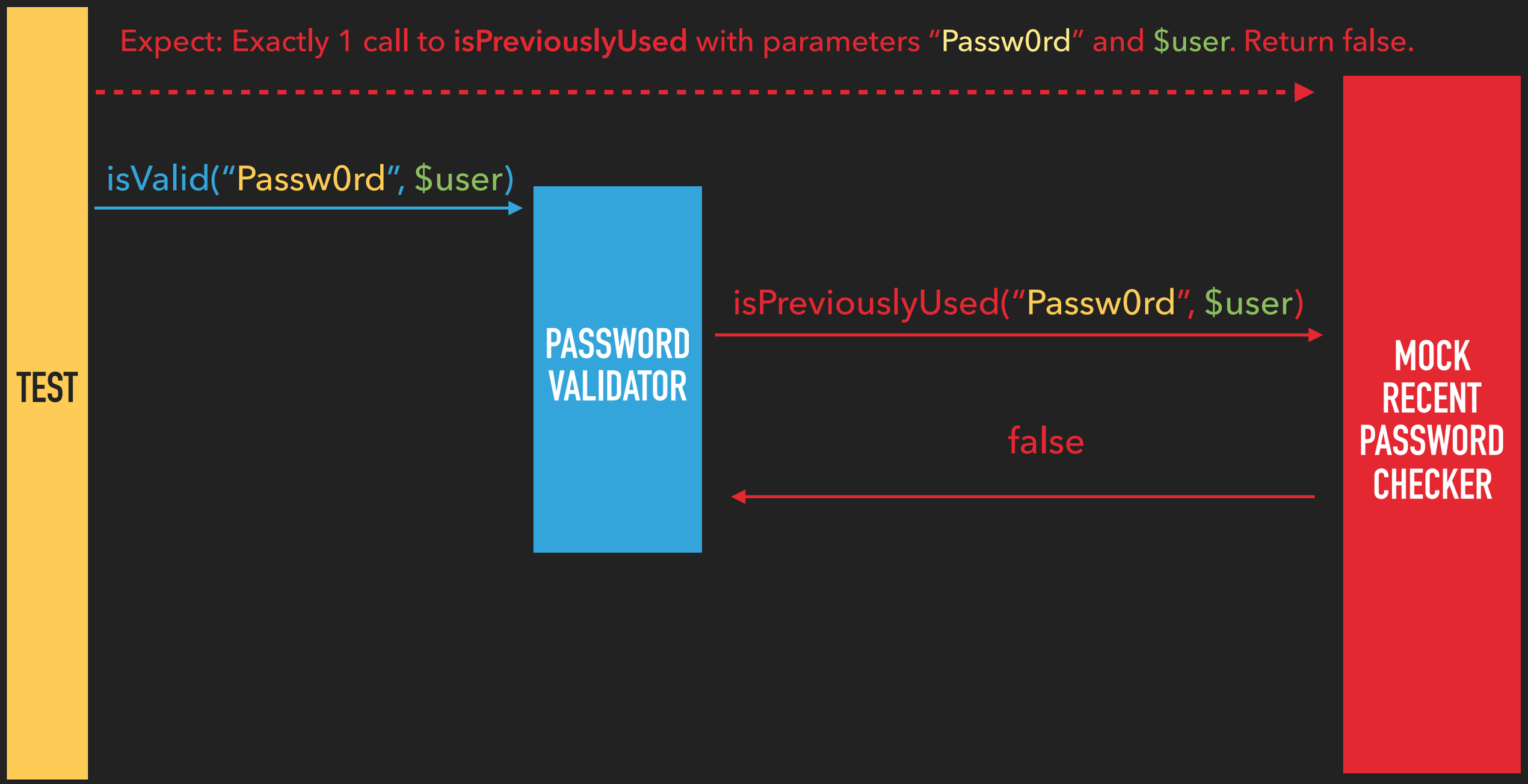
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



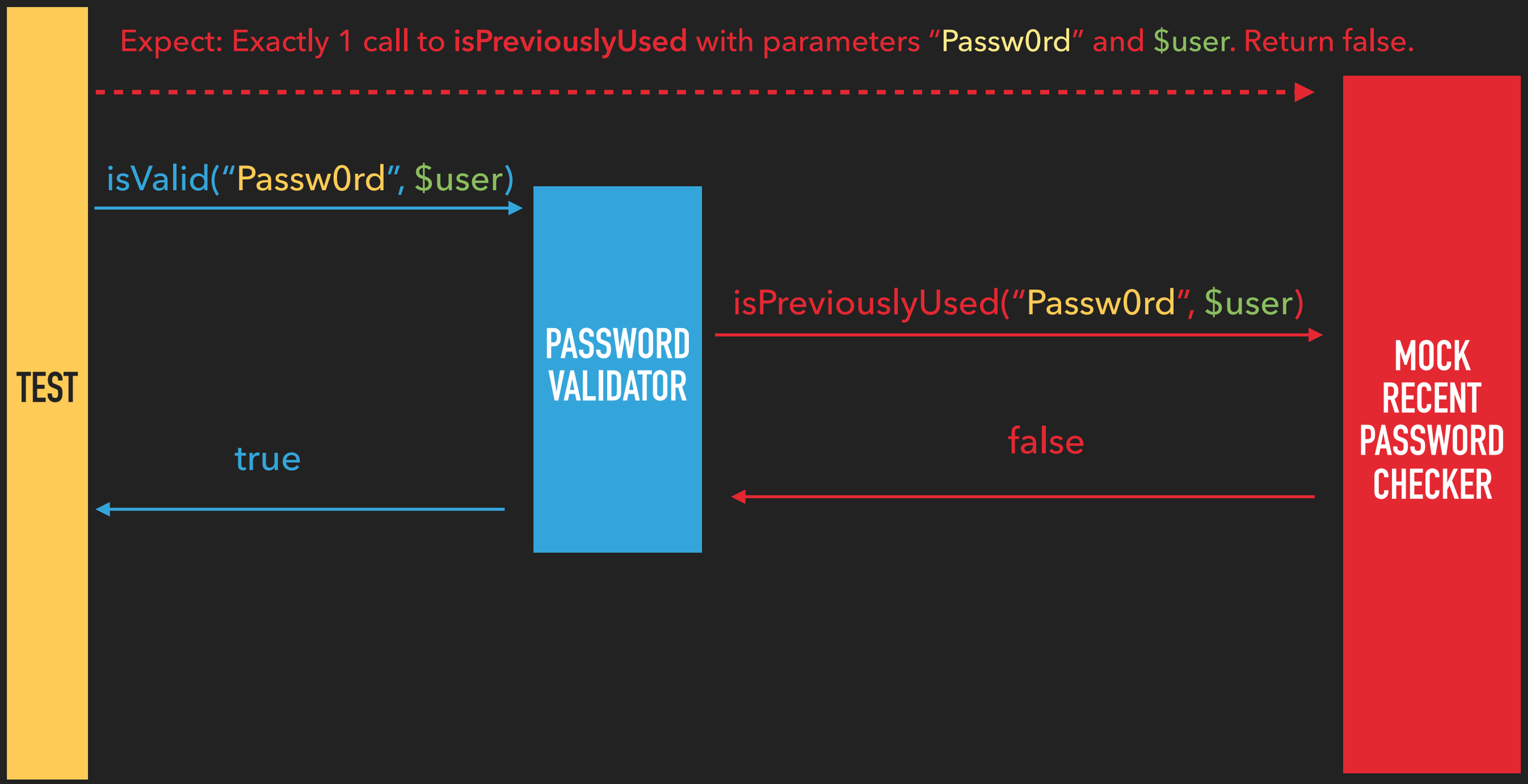
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



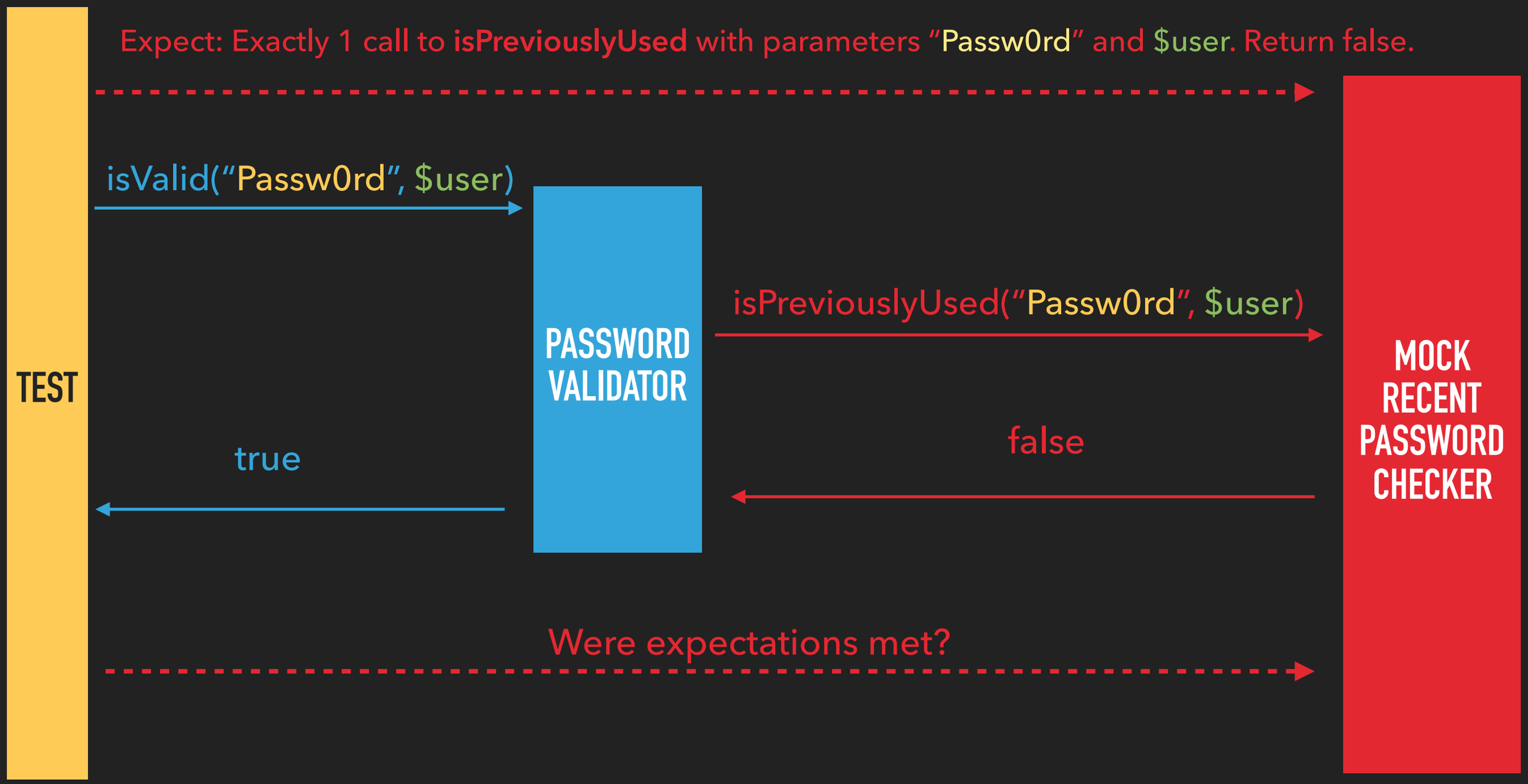
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



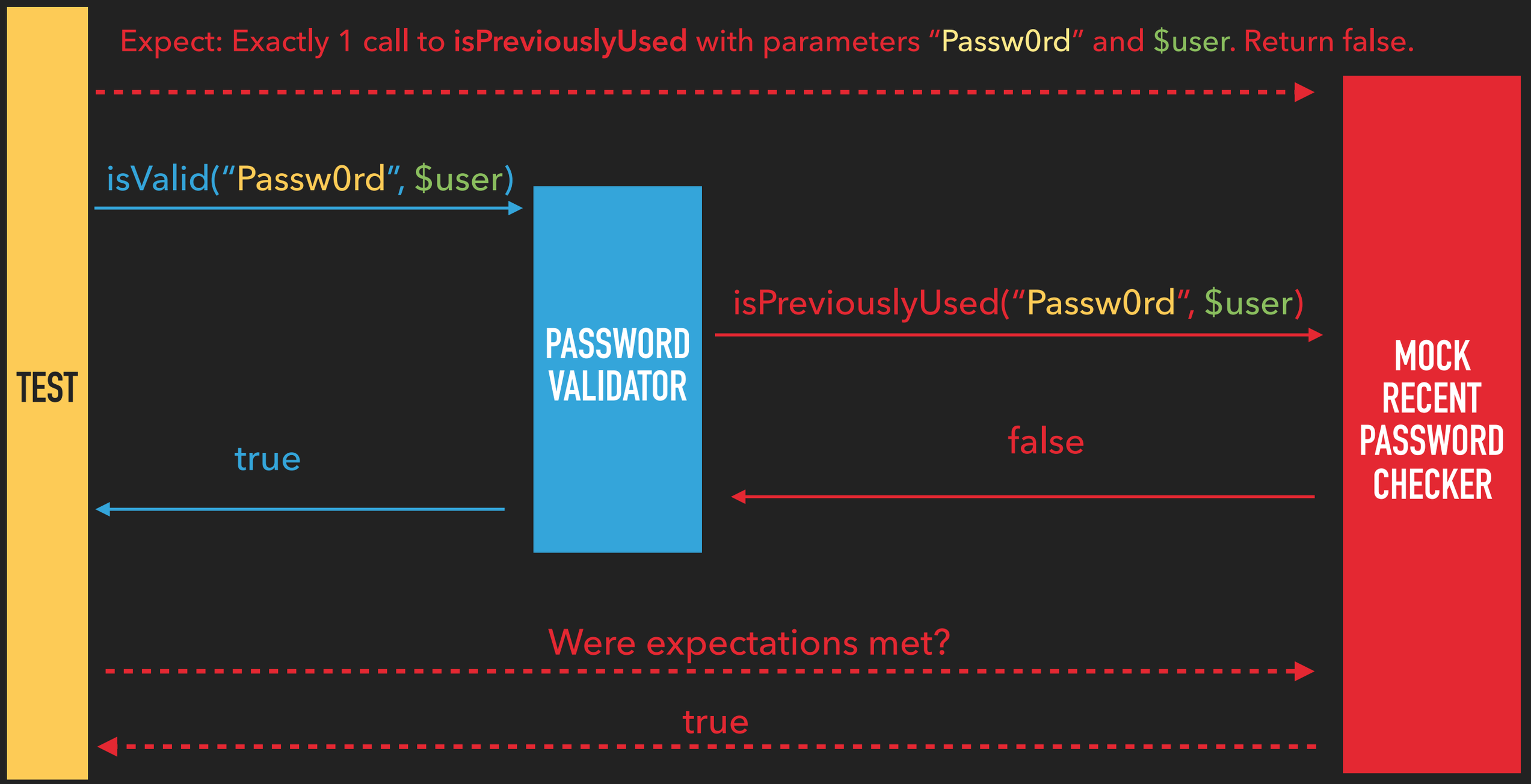
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



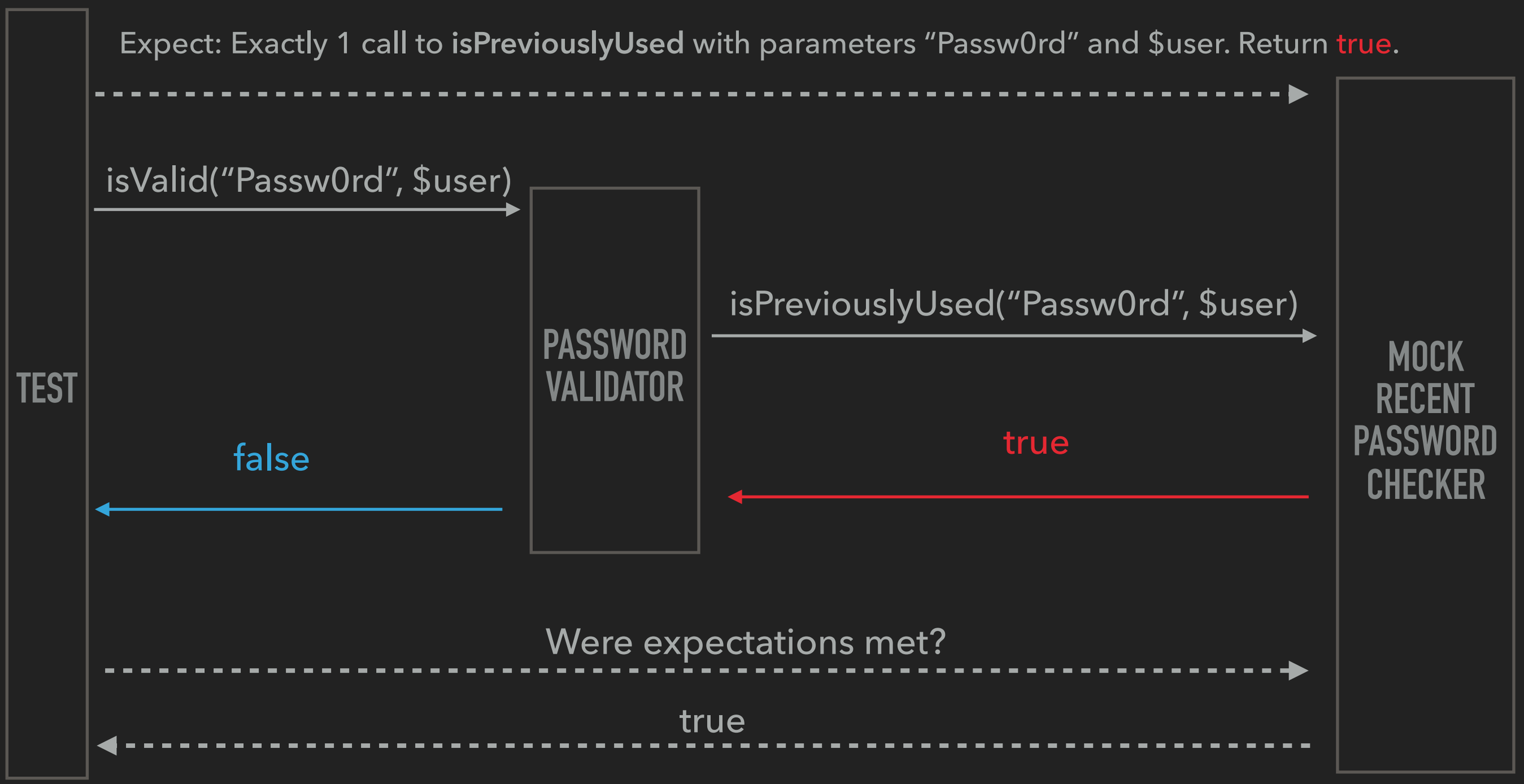
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



NEW TEST: VALID PASSWORD, NOT RECENTLY USED



NEW TEST: VALID PASSWORD, BUT RECENTLY USED



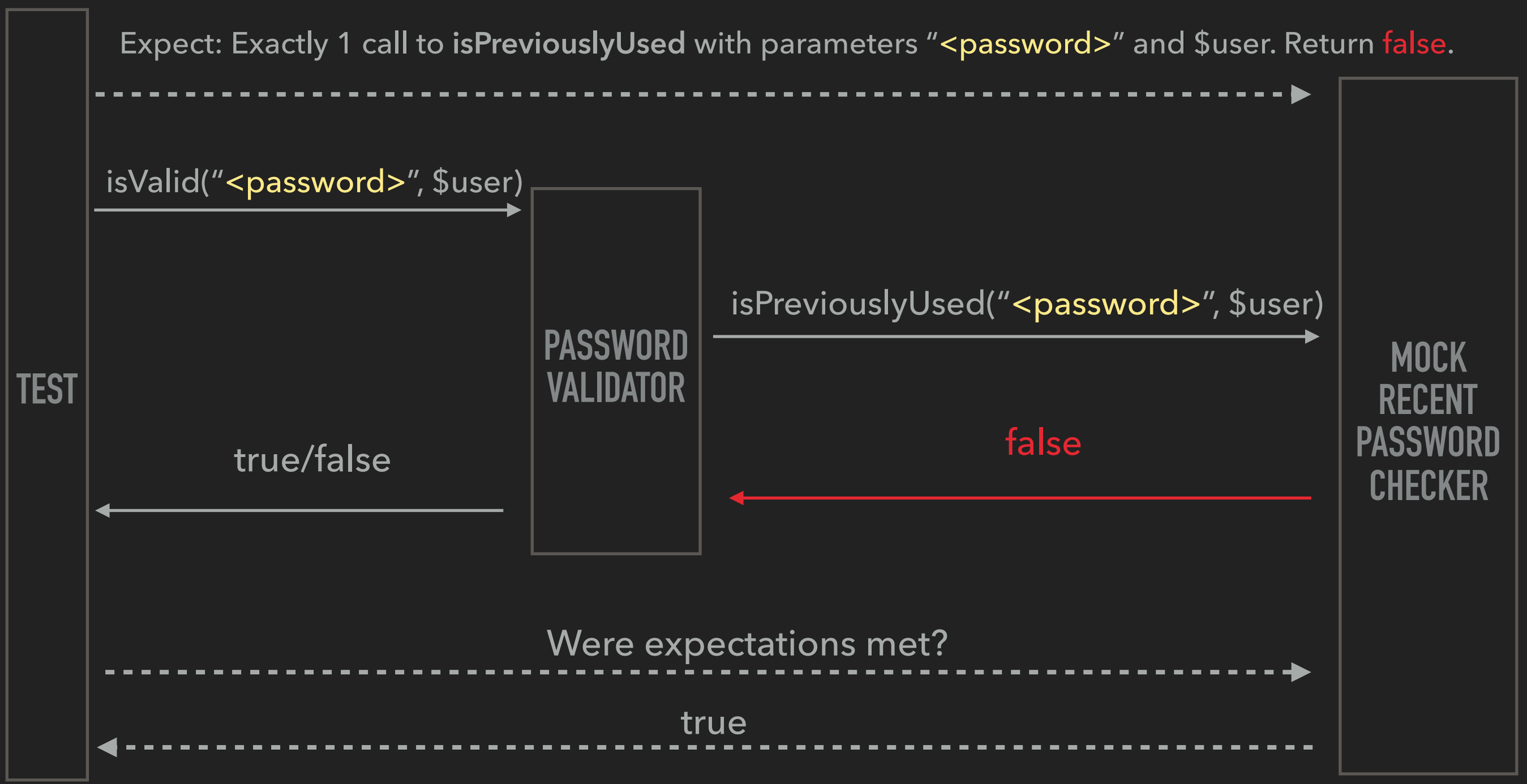
THESE EXTRA 2 TESTS ARE THE AWKWARD DUO

EXISTING TESTS (FAMOUS FIVE)

EXISTING TESTS (FAMOUS FIVE)

```
class PasswordValidator
{
  public function isValid(string $password, User $user) : bool
  {
     if ($this->recentPasswordChecker->isRecentPassword(
             $password, $user)) {
       return false;
    }

    if (… password too short …) return false;
    if (… password has no digit …) return false;

   … remaining checks …

   return true;
}
```

EXISTING TESTS

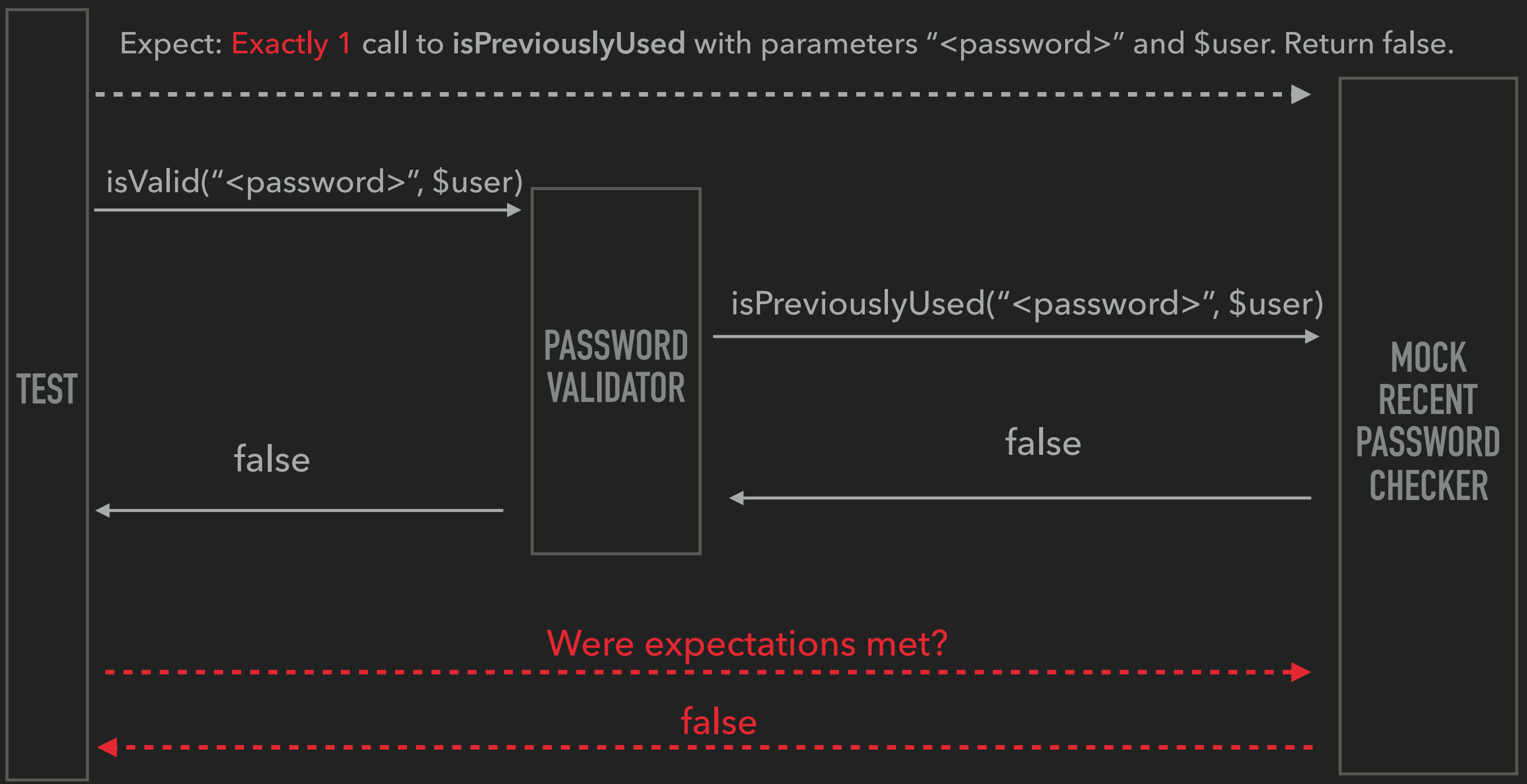


EXISTING TESTS – REFACTOR CODE

```
class PasswordValidator
{
  public function isValid(string $password, User $user) : bool
  {
     if (… password too short …) return false;
     if (… password has no digit …) return false;
```

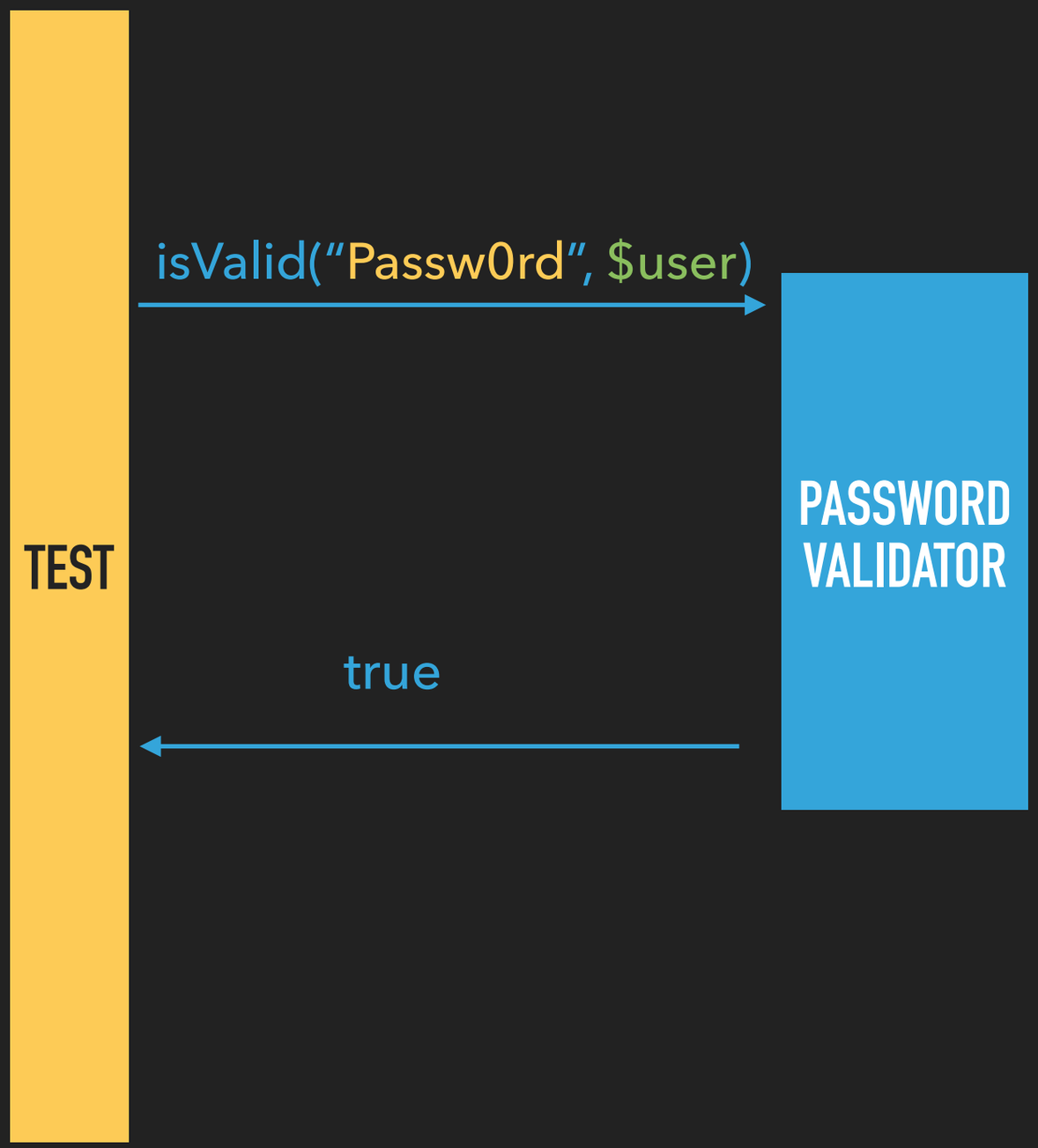
```
   … remaining checks …

    if ($this->recentPasswordChecker->isRecentPassword(
              $password, $user)) {
      return false;
    }

 return true;
}
```

EXISTING TESTS: AFTER REFACTOR

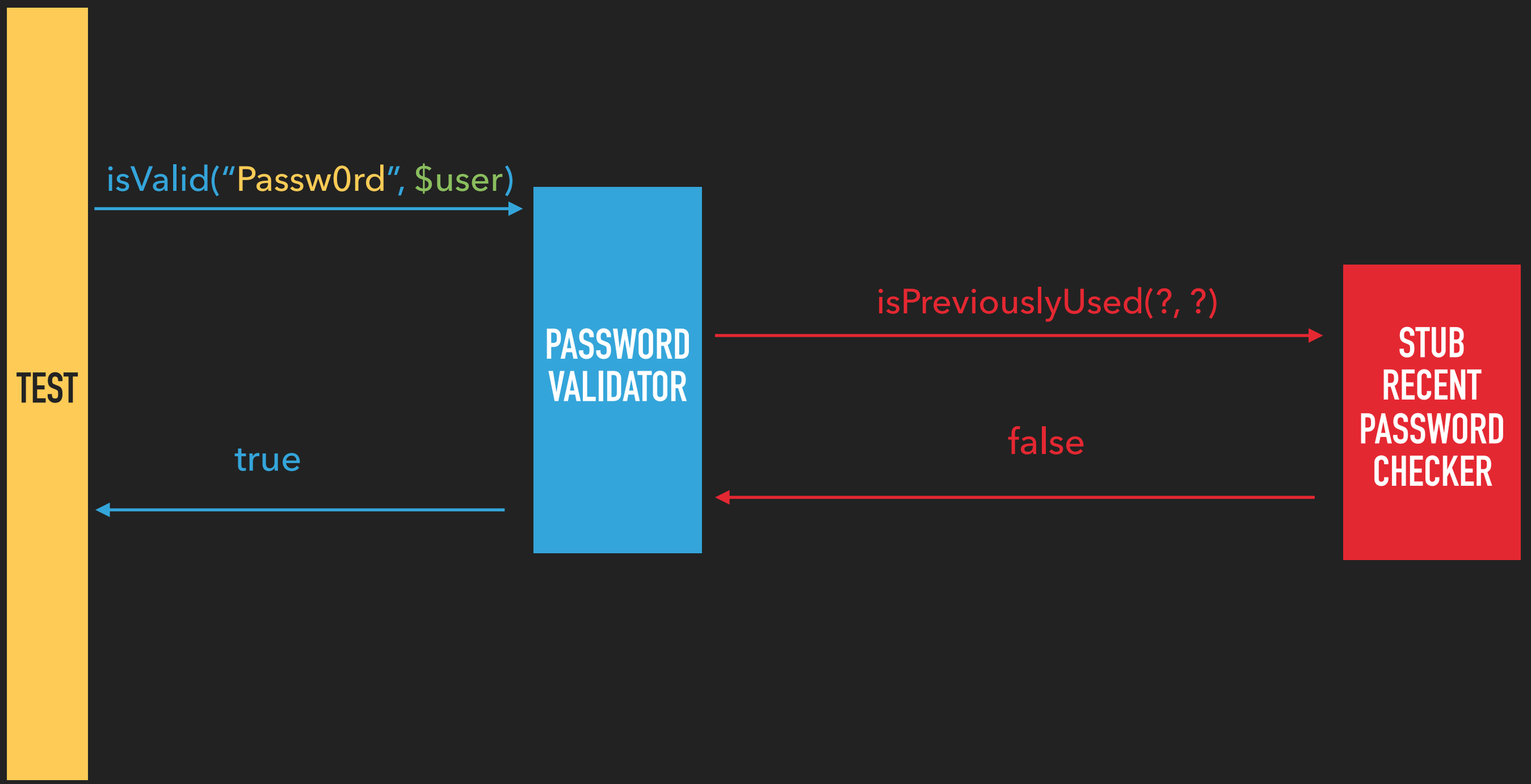


WE'VE REFACTORED CODE AND THE TESTS HAVE BROKEN. NOT GOOD!

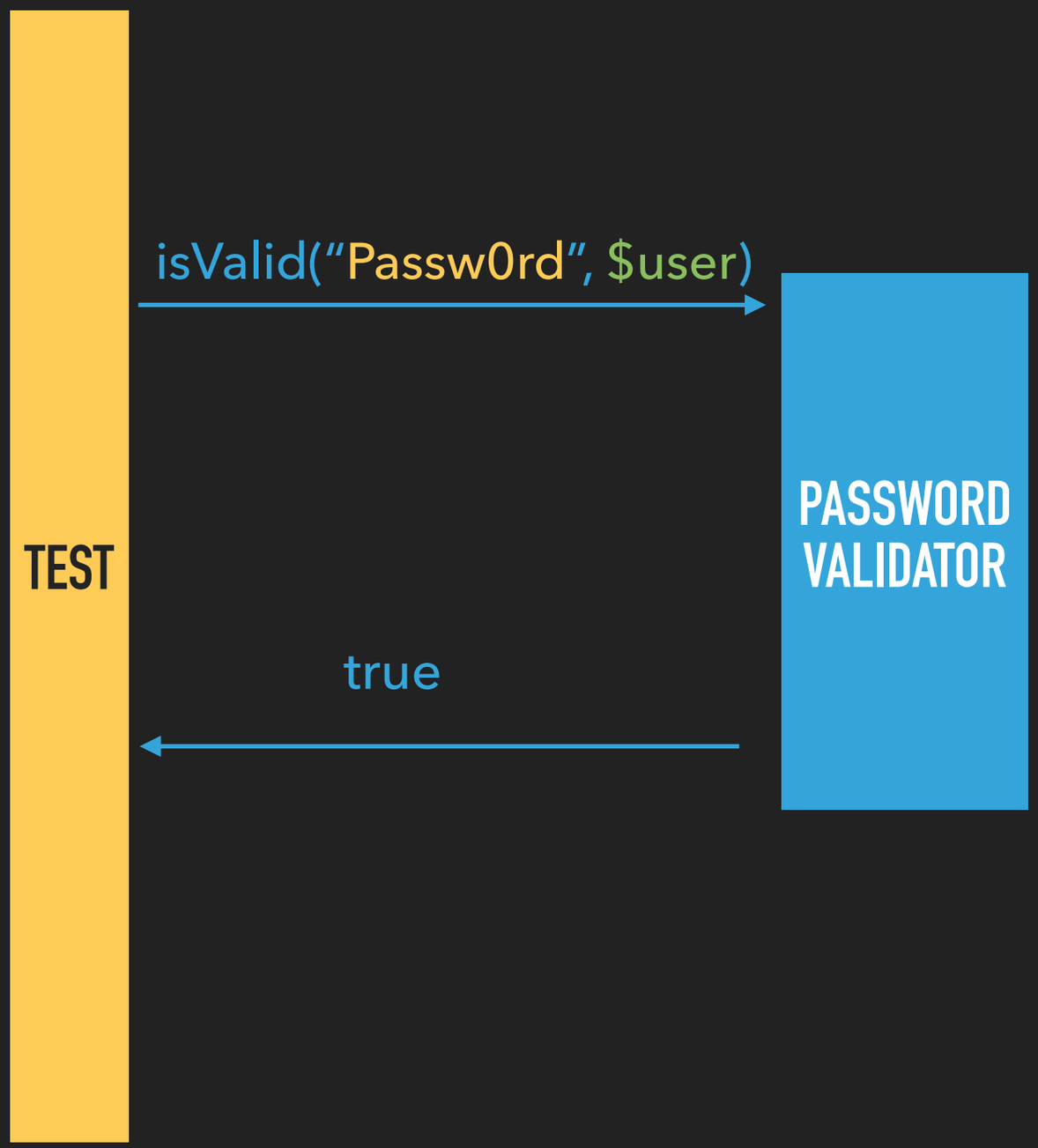
USE A STUB



USE A STUB


isValid(“Passw0rd”, $user)
TEST
PASSWORD VALIDATOR
isPreviouslyUsed(?, ?)
STUB RECENT PASSWORD CHECKER
false
true

USE A STUB



HAND CODE STUB?

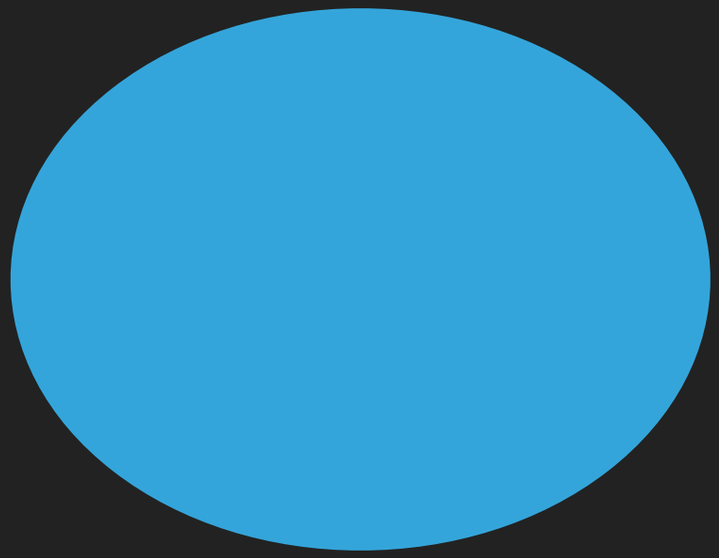
```
StubPasswordChecker implements PreviousPasswordChecker
{

  public function isPreviouslyUsed(
     string $password, User $user) : bool {
     return false;
  }
}
```

TAKE AWAY

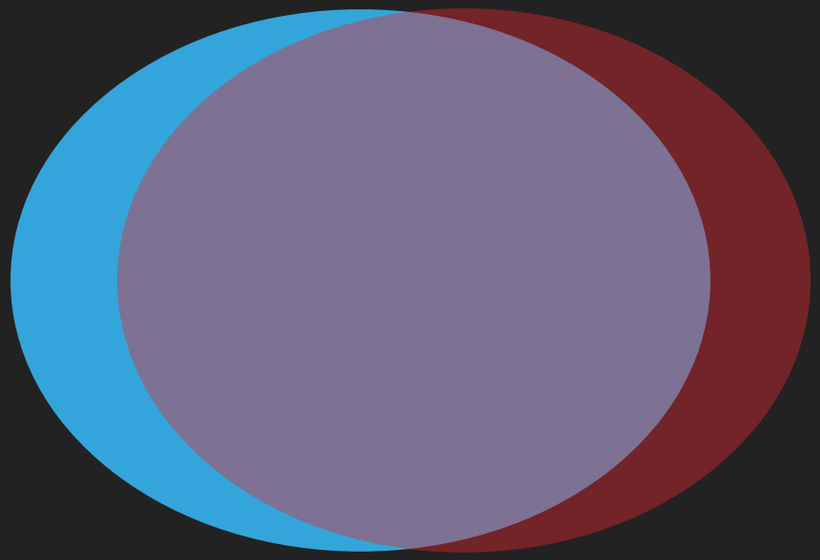
USE STUBS UNLESS YOU REALLY NEED MOCKS

- Limit the coupling between tests and the code
- Unnecessary coupling increases maintenance cost
  - tests harder to write in the first place
  - reduces ability to refactor

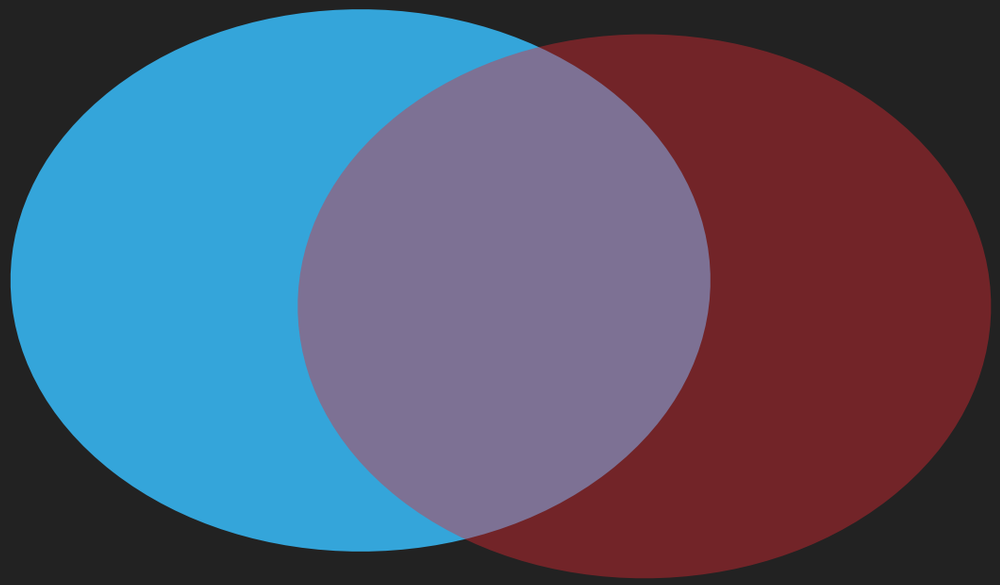
TEST DOUBLE IS AN APPROXIMATION



TEST DOUBLE IS AN APPROXIMATION



TEST DOUBLE IS AN APPROXIMATION



IMPROVE THE INTERFACE

```
interface PreviousPasswordChecker
{

   /**
    * Returns true if password has been used by user
    * in previous 5 passwords
    */
    public function isPreviouslyUsed(
          $password,
          $user
    )

}
```

IMPROVE THE INTERFACE

```
interface PreviousPasswordChecker
{

   /**
    * Returns true if password has been used by user
    * in previous 5 passwords
    */
    public function isPreviouslyUsed(
    string  $password,
            $user
    )

}
```

IMPROVE THE INTERFACE

```
interface PreviousPasswordChecker
{

   /**
    * Returns true if password has been used by user
    * in previous 5 passwords
    */
    public function isPreviouslyUsed(
    string  $password,
    User    $user
    )

}
```

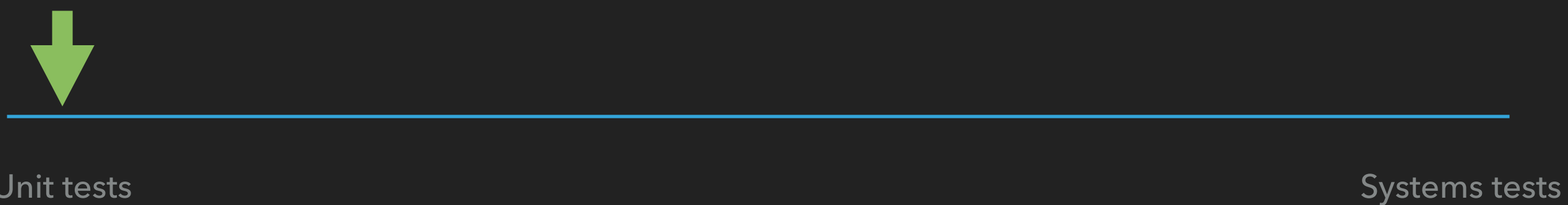
IMPROVE THE INTERFACE

```
interface PreviousPasswordChecker
{

    /**
     * Returns true if password has been used by user
     * in previous 5 passwords
     */
    public function isPreviouslyUsed(
    string  $password,
    User    $user
    ) : bool

}
```

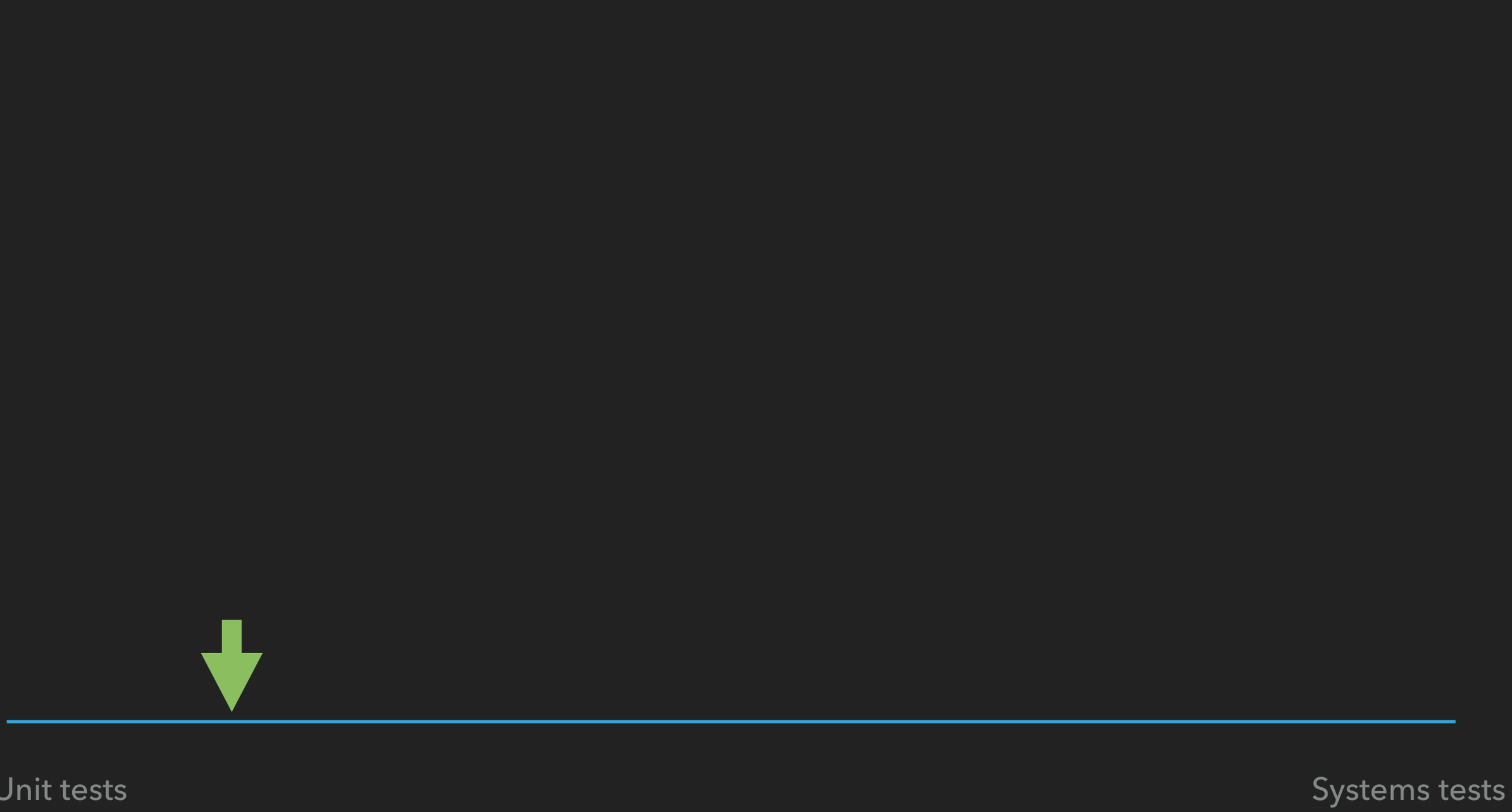
OTHER REASONS FOR DIFFERENCES BETWEEN TEST DOUBLE

- Specification might change
- Specification might be misunderstood
- Functionality might not be implemented

WHERE ALONG THE CONTINUUM ARE WE NOW



WHERE ALONG THE CONTINUUM ARE WE NOW

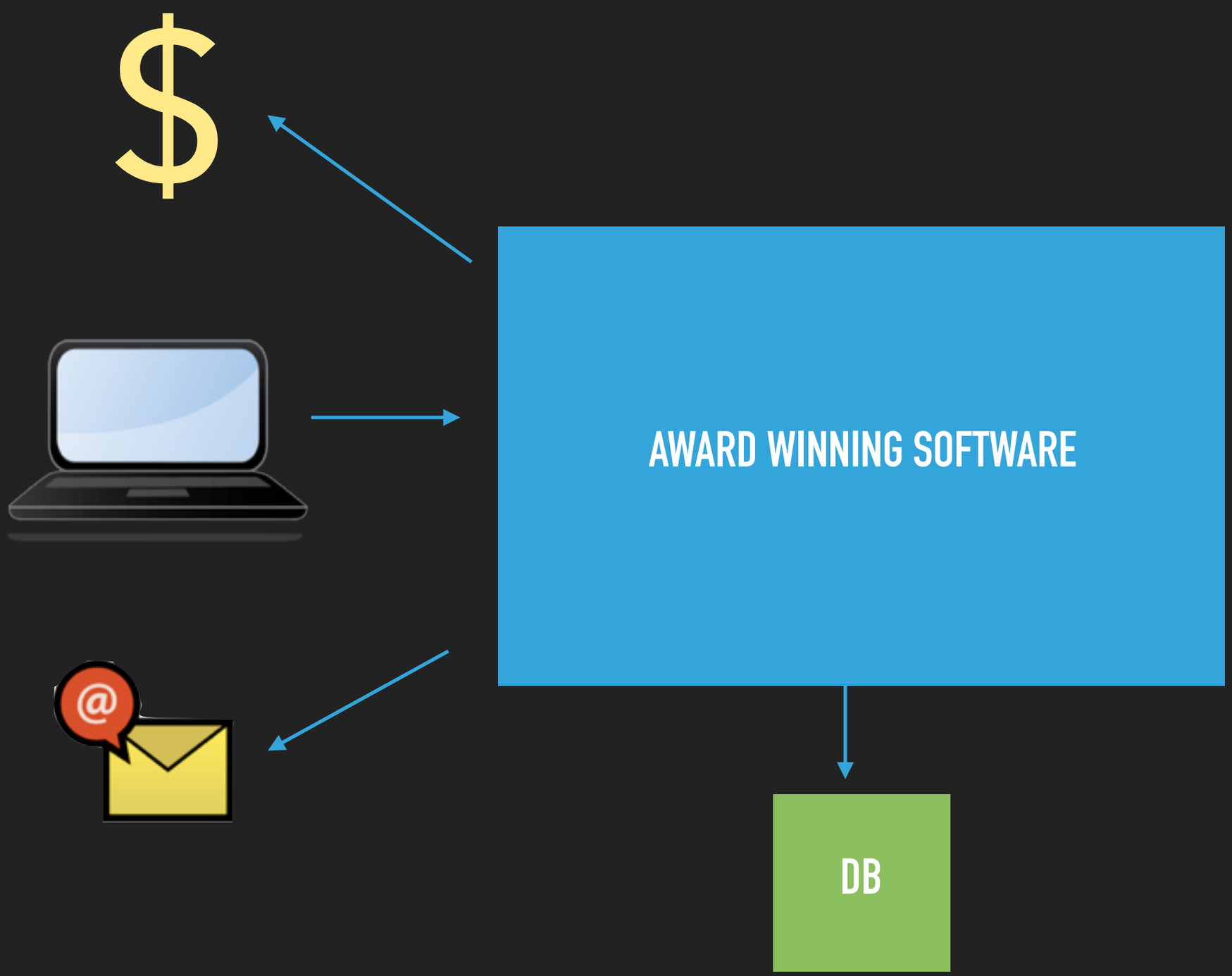


UNIT TEST LEVEL

- Good architecture makes testing easier
  - Examples of S L I D of SOLID
- Decouple tests from code as much as possible
  - E.g. use stubs unless you really need a mock

BIGGER TESTS

ARCHITECTURE



ARCHITECTURE

CODE TALKS TO EXTERNAL SERVICE

INTERFACE TO EXTERNAL SERVICE

AWARD WINNING SOFTWARE

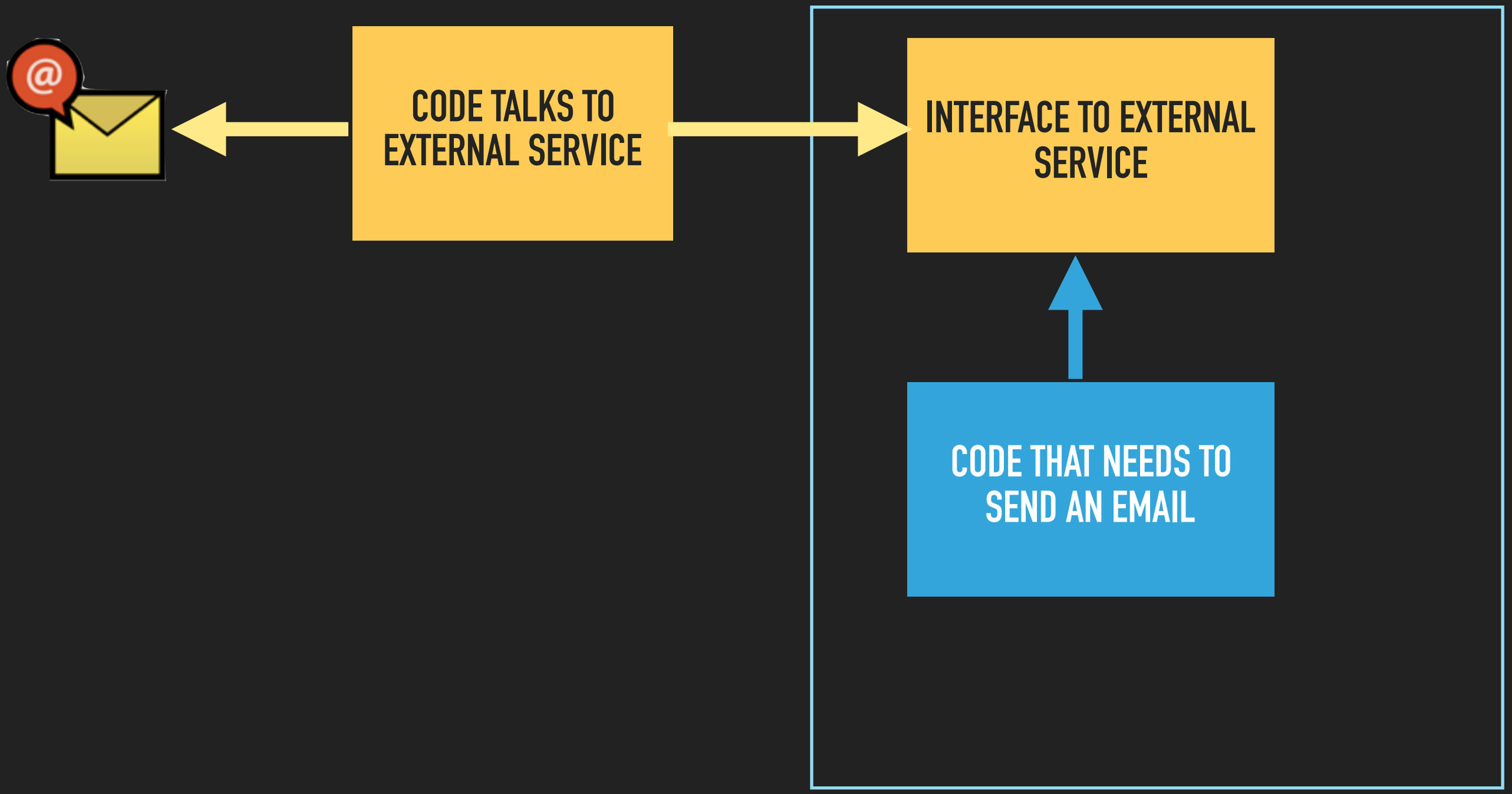
EMAIL GATEWAY INTERFACE

```
interface EmailGatewayInterface
{
    public function sendEmail(EmailMessage $message);

}
```

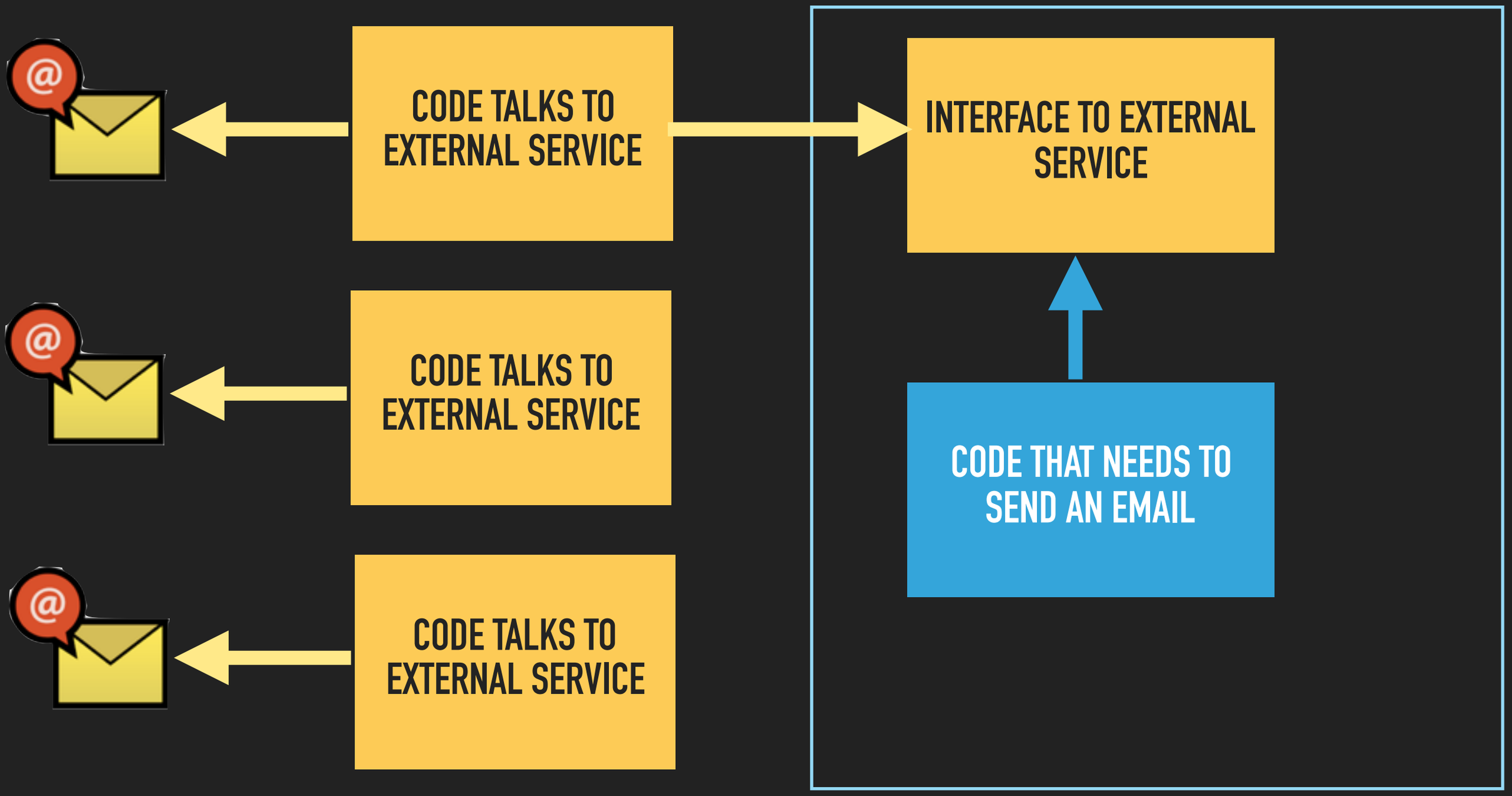
EMAIL MESSAGE OBJECT

- To
- From
- CC
- Subject
- Template
- Data

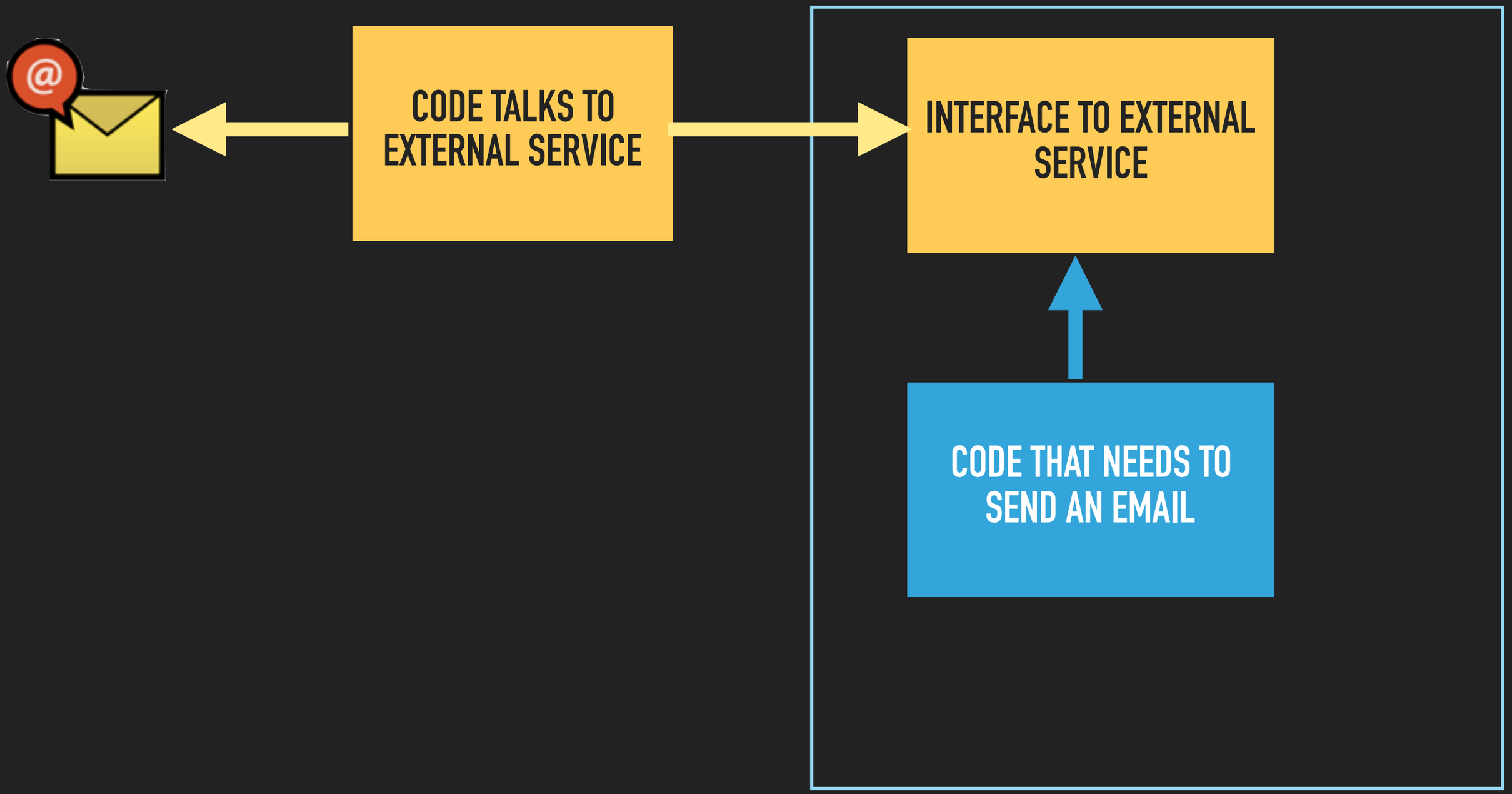
ARCHITECTURE



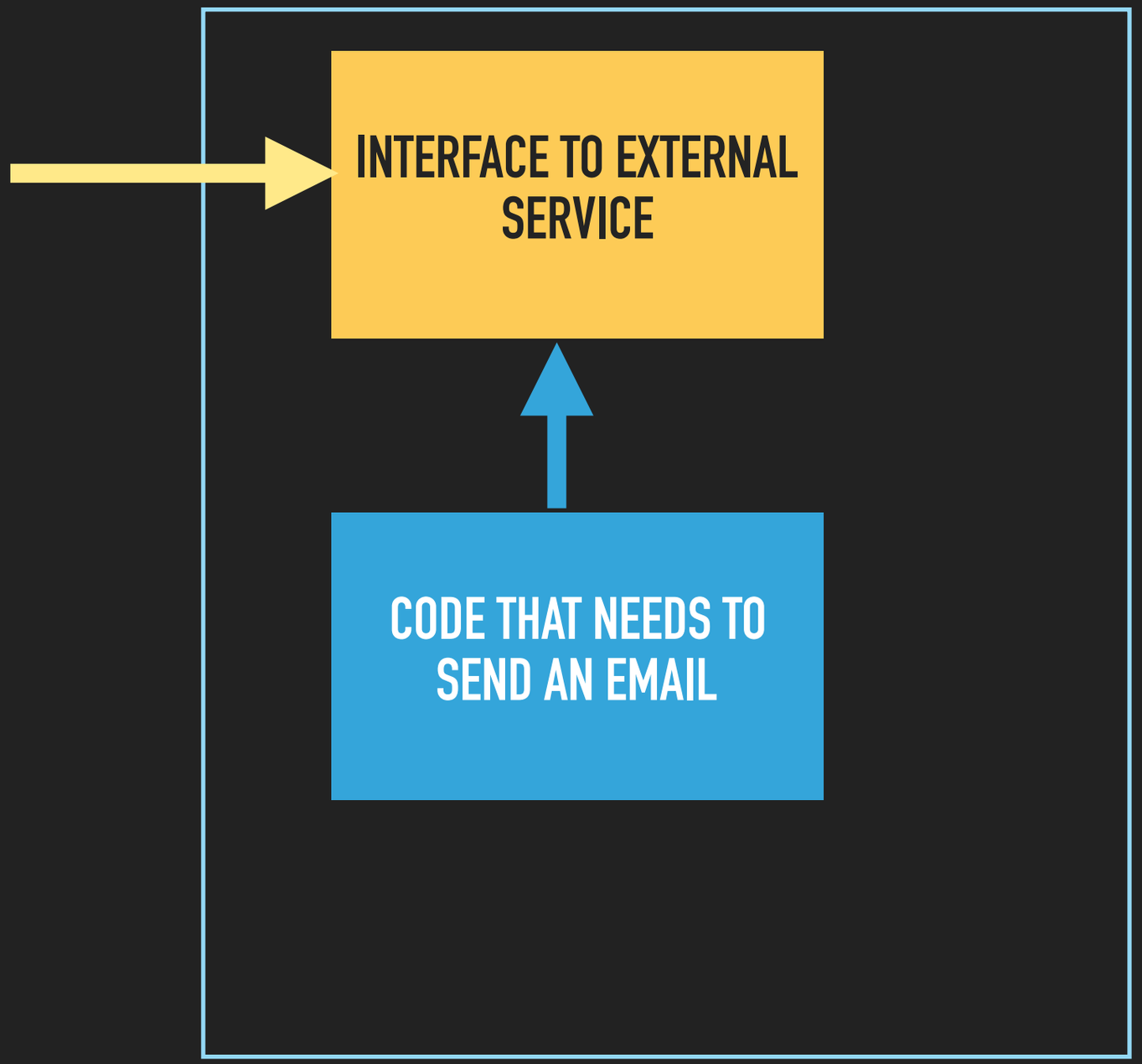
ARCHITECTURE



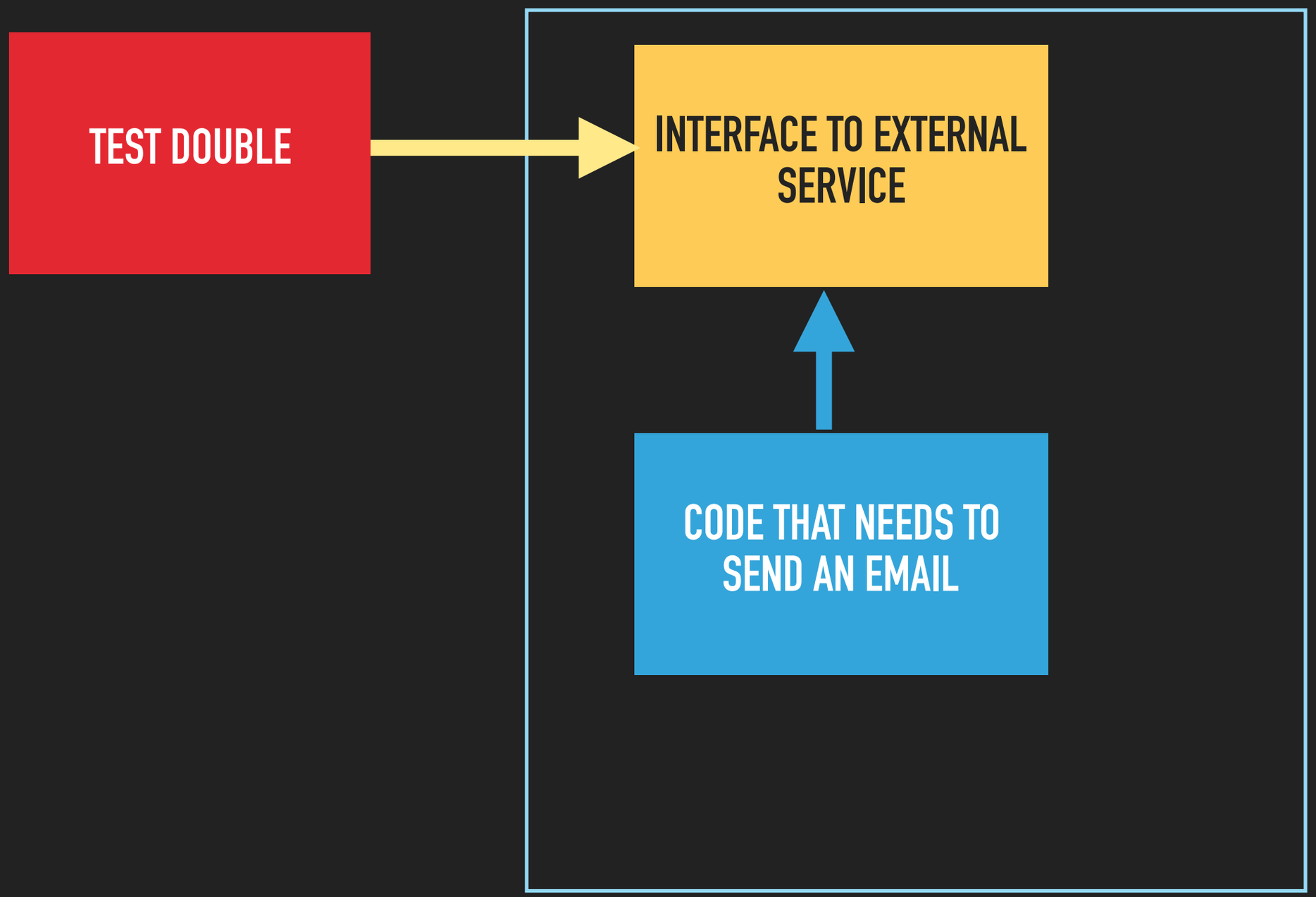
ARCHITECTURE



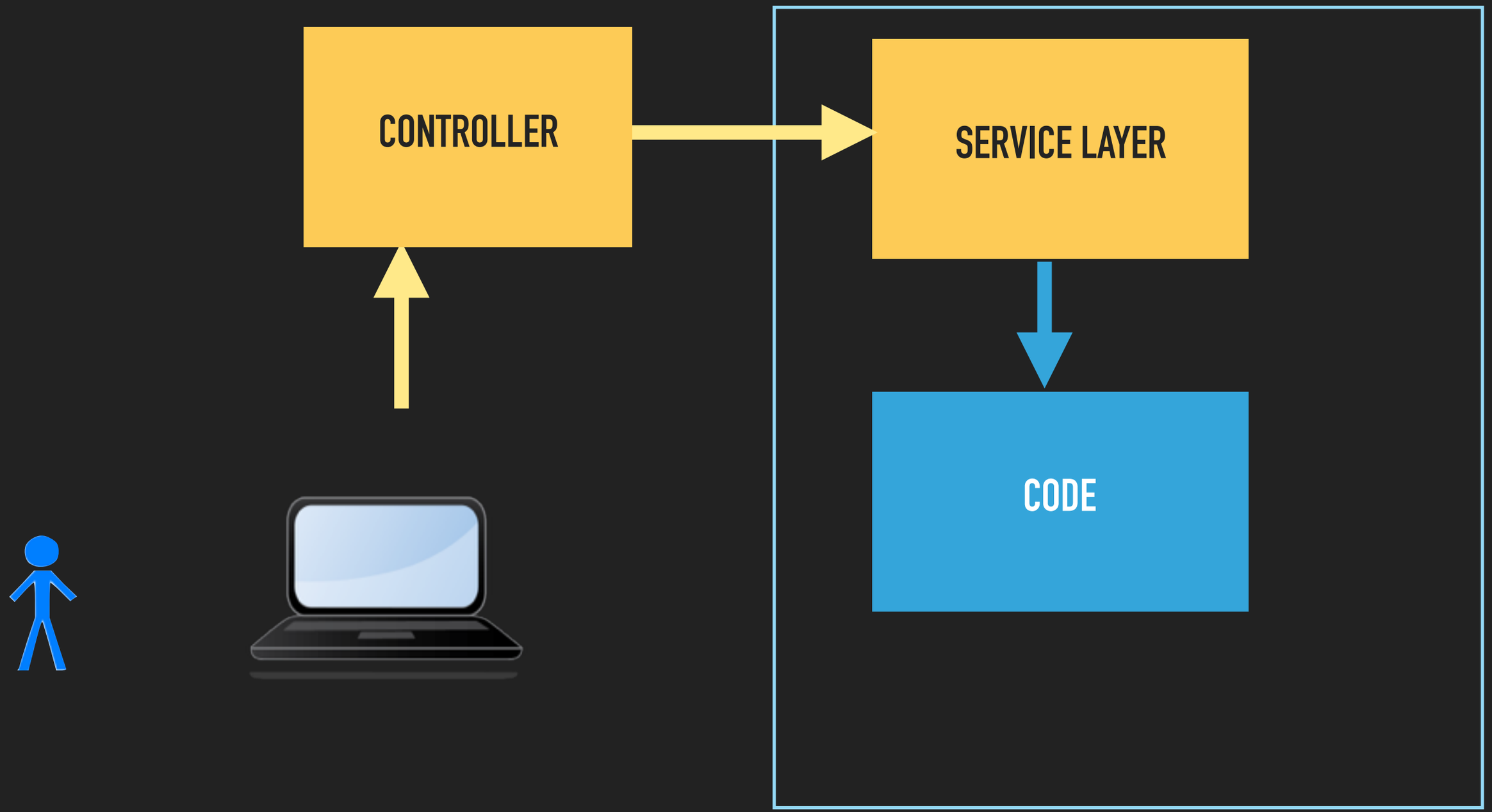
ARCHITECTURE

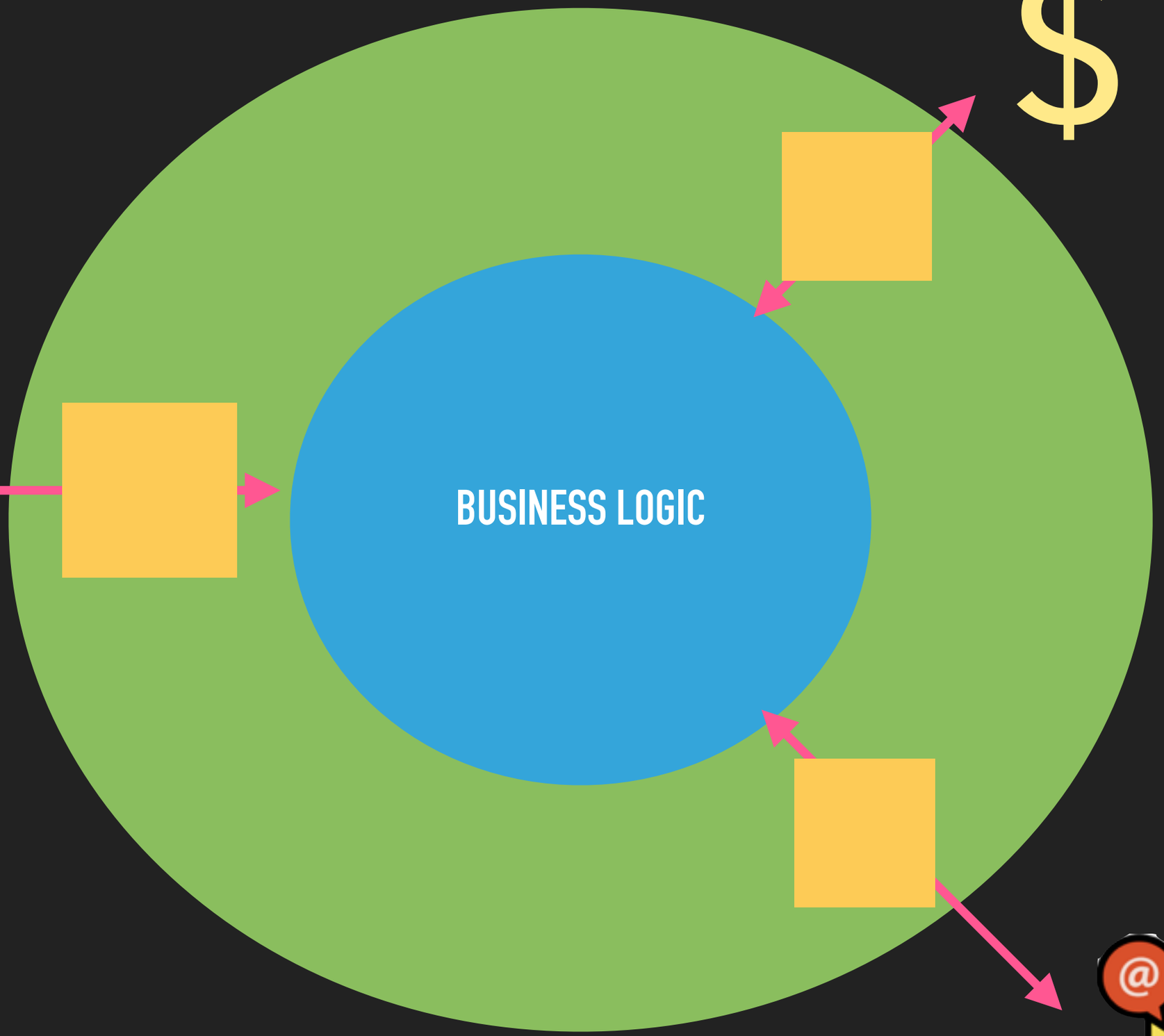


ARCHITECTURE



ARCHITECTURE




$
BUSINESS LOGIC

ARCHITECTURE

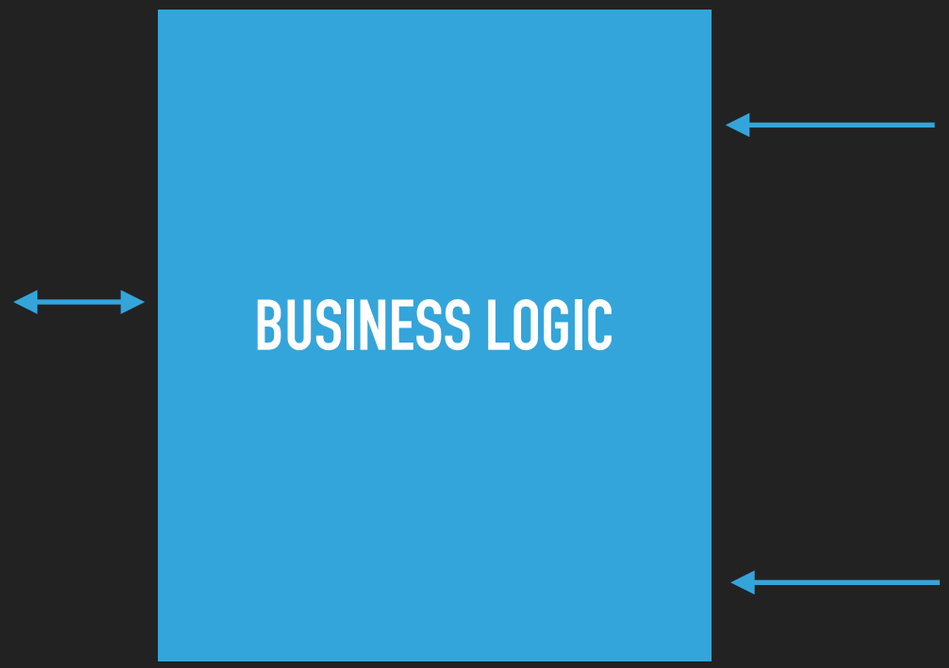
CONTROLLER

BUSINESS LOGIC

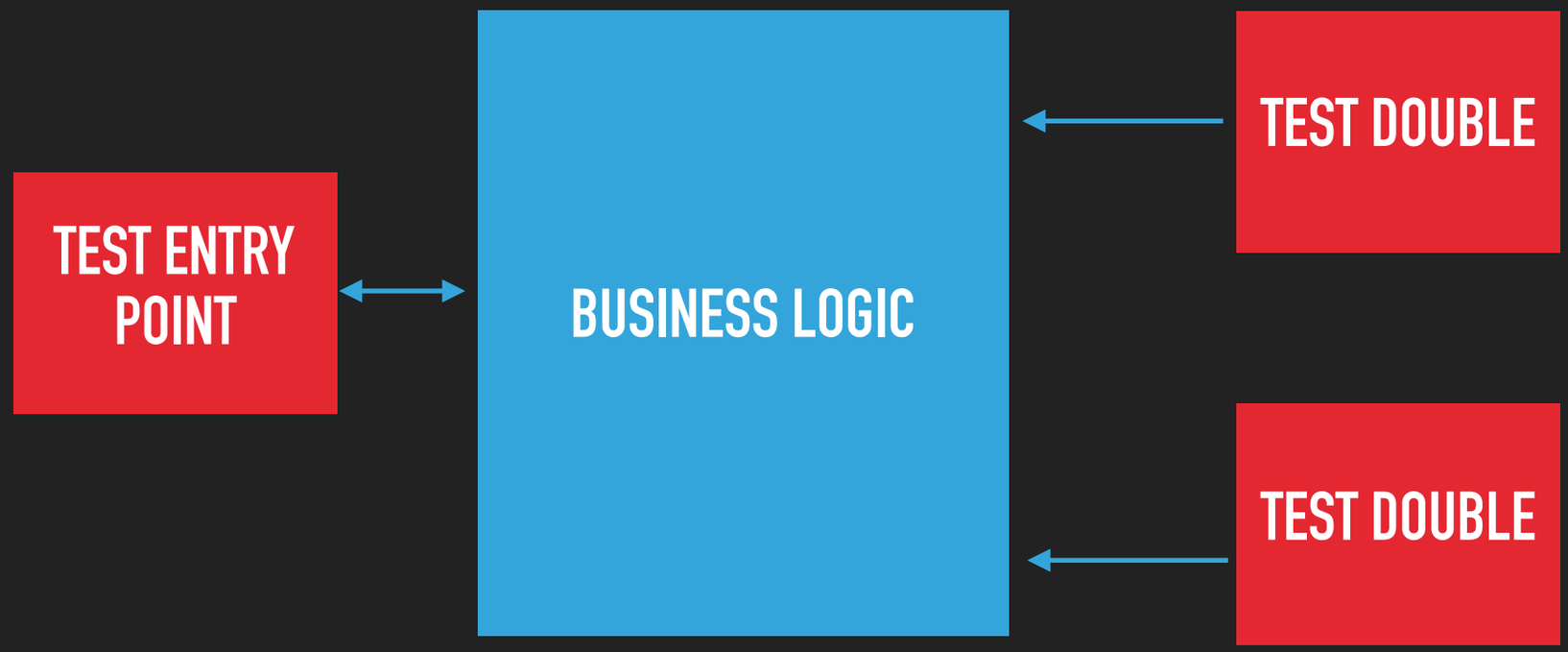
PAYMENT GATEWAY

EMAIL GATEWAY

ARCHITECTURE



ARCHITECTURE



FAKE EMAIL GATEWAY

```
class FakeEmailGateway implements EmailGatewayInterface
{

  private $emailMessages = [];

  public function sendEmail(EmailMessage $message){
    $this->emailMessages[] = $message;
  }


  public function findBy($to, $template): array {
    … return EmailMessage meeting criteria …
  }
}
```

REGISTER USER TEST 1

```
class RegisterUserTest extends TestCase
{

  public function testRegistration() {

    $userService = $this->container->getUserService();

$success = $userService->registerUser(
      “dave@lampbristol.com”, “Dave”, “Passw0rd”);

$this->assertTrue($success);


…
```

REGISTER USER TEST 2

```
…
  $emailGateway = $this->container->getEmailGateway();

   $emailMessages = $emailGateway->find(
      “dave@lampbristol.com”, “REGISTRATION”);

  $this->assertCount(1, $emailMessages);

  $data = $emailMessage->getData();
  $confirmationToken = $data[‘confirmationToken’];

  $success = $userService->completeRegistration(
      $confirmationToken);

  $this->assertTrue($success);
```

TAKE AWAY

ARCHITECTURE IS VERY IMPORTANT

- Apply SOLID principles to your codebase
- A code base isn't difficult to test, it's poorly architected.

RETURNING TO OUR PASSWORD VALIDATOR: 1

```
class PasswordValidatorTest extends TestCase
{

  public function testUpdatePassword() {

    … create $user with password ‘Passw1rd’ …

    $userService = $this->container->getUserService();

    $userService->updatePassword($user, ‘Passw2rd’);
    $userService->updatePassword($user, ‘Passw3rd’);
    $userService->updatePassword($user, ‘Passw4rd’);

…
```

RETURNING TO OUR PASSWORD VALIDATOR: 2

…

```
$success = $userService->updatePassword(
        $user, 'Passw1rd');

$this->assertFalse($success);
```

```
$success = $userService->updatePassword(
        $user, 'Passw5rd');

$this->assertTrue($success);
```

CONSTRUCTING OUR USER OBJECT

```
… create $user with password ‘Passw1rd’ …
```

HOW DO WE BUILD THE TEST USER OBJECT

- Hand build what is required
- Seed the database
- Object mother
- Test Builder

HAND BUILDING

- Fragile
  - What happens if construction method changes?
- Breaks DRY

SEEDING A DATABASE

- OK for very simple data
- Coupling our test data to our database schema design
- Not good for complex data structures
- Not the way the data really got into the system
- Fragile
  - What happens if the construction method changes

OBJECT MOTHER

- Factory
- Multiple
  - UserObjectMother, ProductObjectMother, etc
- Create objects in know states ready for testing:
  - UserObject::createJohn()
    - Normal user
    - Password: Passw1rd

OBJECT MOTHER BENEFITS

- Single place where test business object built
  - Easy to find
  - Easy to update
- Defer to other Object Mothers
- Parameterised to update how object is built:
  - getJohn(“Passw2rd”)

USING AN OBJECT MOTHER

```
… create $user with password ‘Passw1rd’ …
```

```
$userObjectMother = $this->container->getUserObjectMother();

$user = $userObjectMother->getJohn();

… test as before …
```

TEST BUILDER

- Offers all benefits of Object Mother
- Gives more control to alter elements of object.

USING A TEST BUILDER (1)

```
$userBuilder = new UserBuilder();
$user = $userBuilder->build();

// User will have default values for name, email, etc
```

USING A TEST BUILDER (2)

```
$userBuilder = new UserBuilder();
$user = $userBuilder
            ->name(“John”)
            ->email(“John@example.com”)
            ->password(“Passw4rd”)
            ->previousPasswords([
                “Passw1rd”,
                “Passw2rd”,
                “Passw3rd”,
             ])
            ->build();
```

TAKE AWAY

USE OBJECT MOTHER AND TEST BUILDER PATTERNS

- Help make your tests more resilient to change
  - Lowers maintenance cost
  - Increases our coverage

FAMOUS 5 VS AWKWARD DUO?

ASSESS VALUE OF TESTS.
REMOVE ONES THAT ARE DUPLICATED (AND OFFER NO BENEFIT)

CAN WE AUTOMATE ANYTHING ELSE?

```
php bin/console test:emailgateway --to dave@lampbristol.com
Sending email:
To       [dave@lampbristol.com]
From     [test@lampbristol.com]
CC       [dave+1@lampbristol.com]
Subject [Test email 2016-02-08 19:37]
Body     [Hi,
          This is a test email.
          Sent at 2016-02-08 19:37.
          From your tester]
```

WHY DO WE NEED A TEST SUITE

- Prove code works
- Prevent against regression
- Allow safe refactoring of code

OUR IDEAL TEST SUITE WOULD BE...

- Fast to execute
- High coverage
- Low maintenance

EVERY THING IS A COMPROMISE

- Nothing is black and white

TO MAKE A GOOD TEST SUITE

- Requires skill
- Good code architecture
- Reduce coupling between tests and code under test:
  - Use mocks only when needed
  - Use patterns like Object Mother and Test Builder

QUESTIONS

ARCHITECTURE