DAVE LIDDAMENT

TEST SUITE HOLY TRINITY

LET'S START WITH A STORY...

WHY ARE WE HERE

- What went wrong
- Why testing will help
- Why good architecture is essential.
- How can we build a good test suite

Dave Liddament

@daveliddament

Lamp Bristol

Organise PHP-SW and Bristol PHP Training

BACK TO THE NIGHTMARE....

WE NEED A TEST SUITE

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

EVERYTHING IS COMPROMISE

TERMINOLOGY

TESTING CONTINUUM

TESTING CONTINUUM

Unit tests

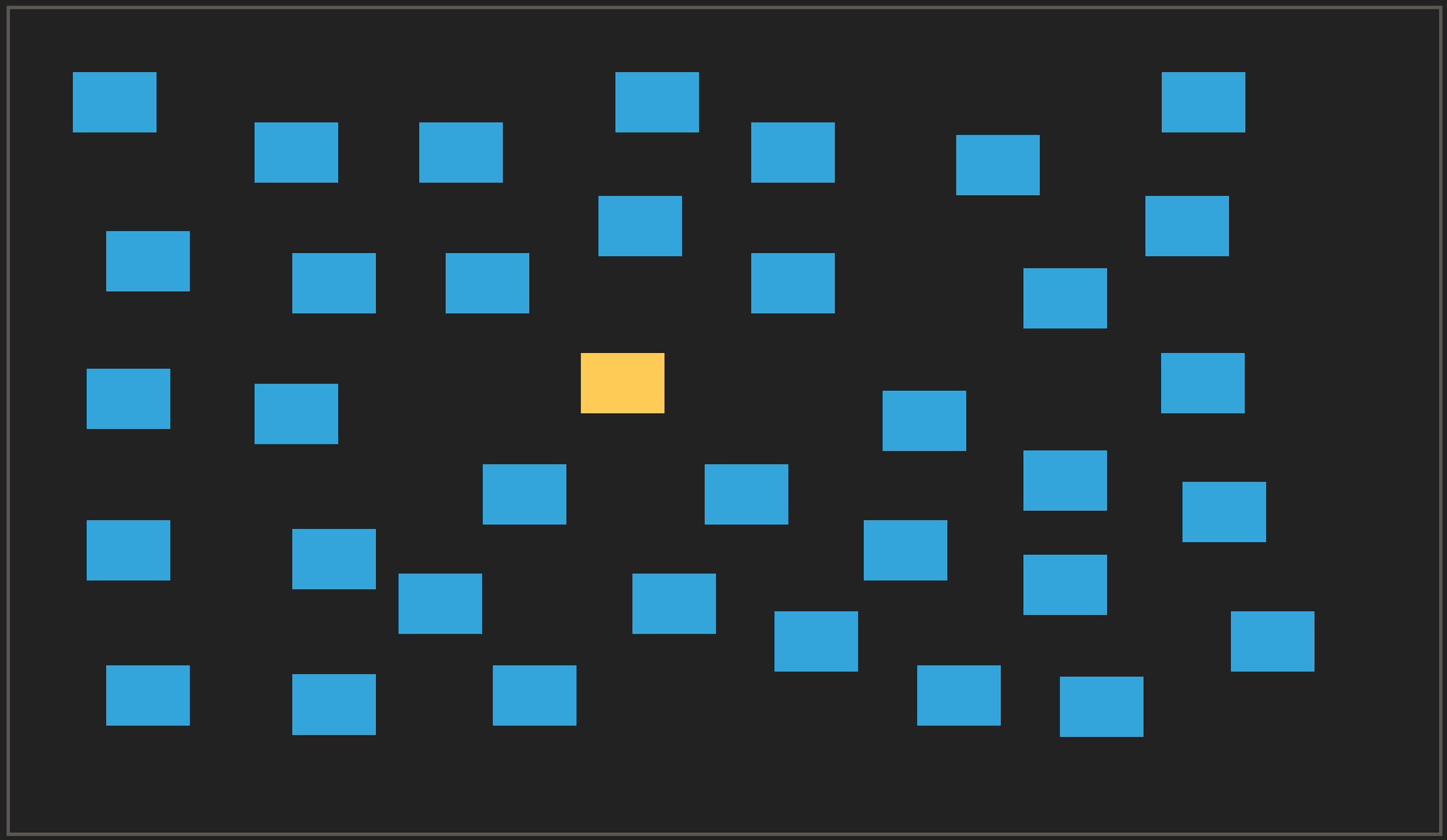
Systems tests

SYSTEM TEST

Unit tests

Systems tests

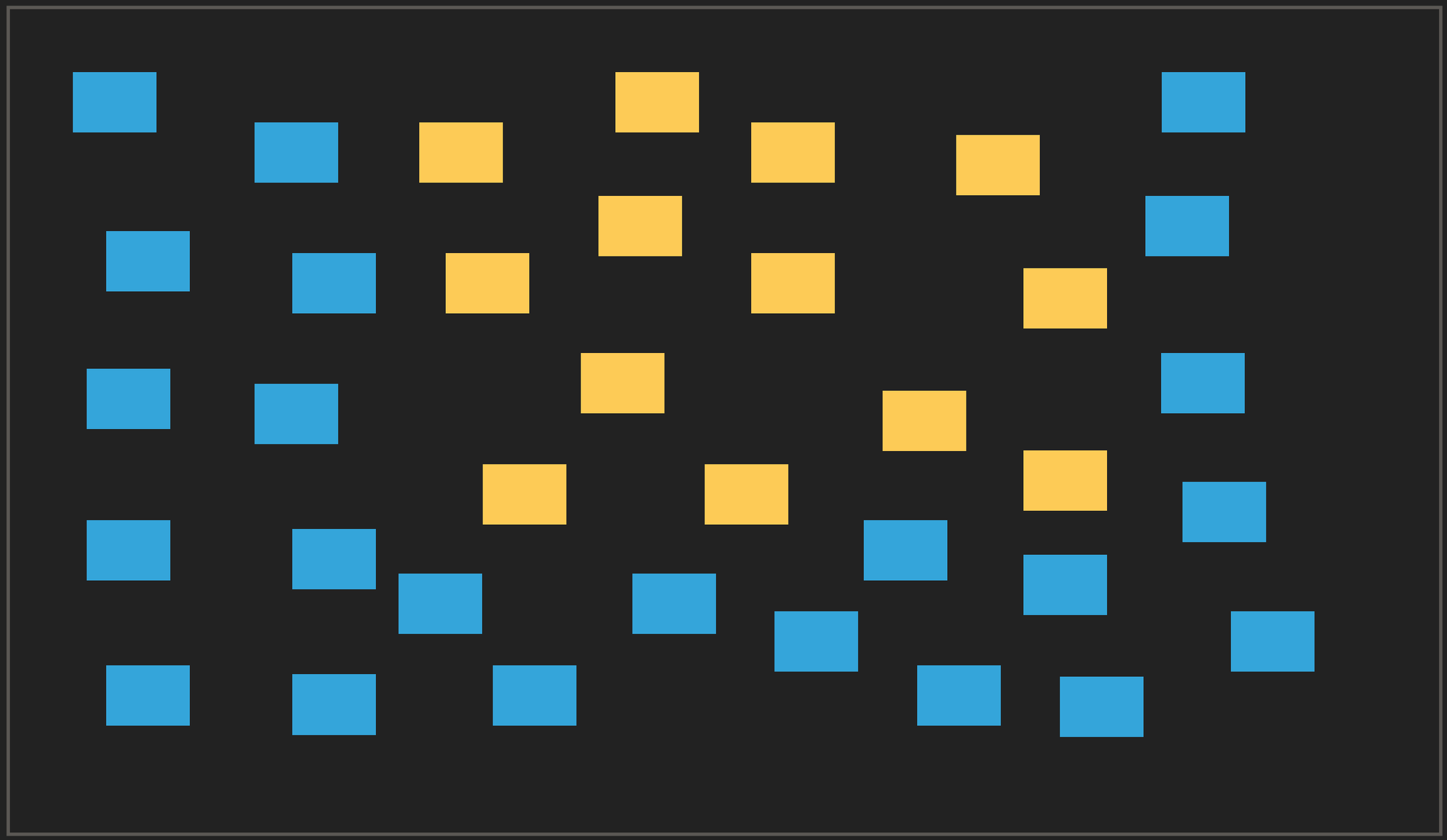
TESTING CONTINUUM



Unit tests

Systems tests

TESTING CONTINUUM



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

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

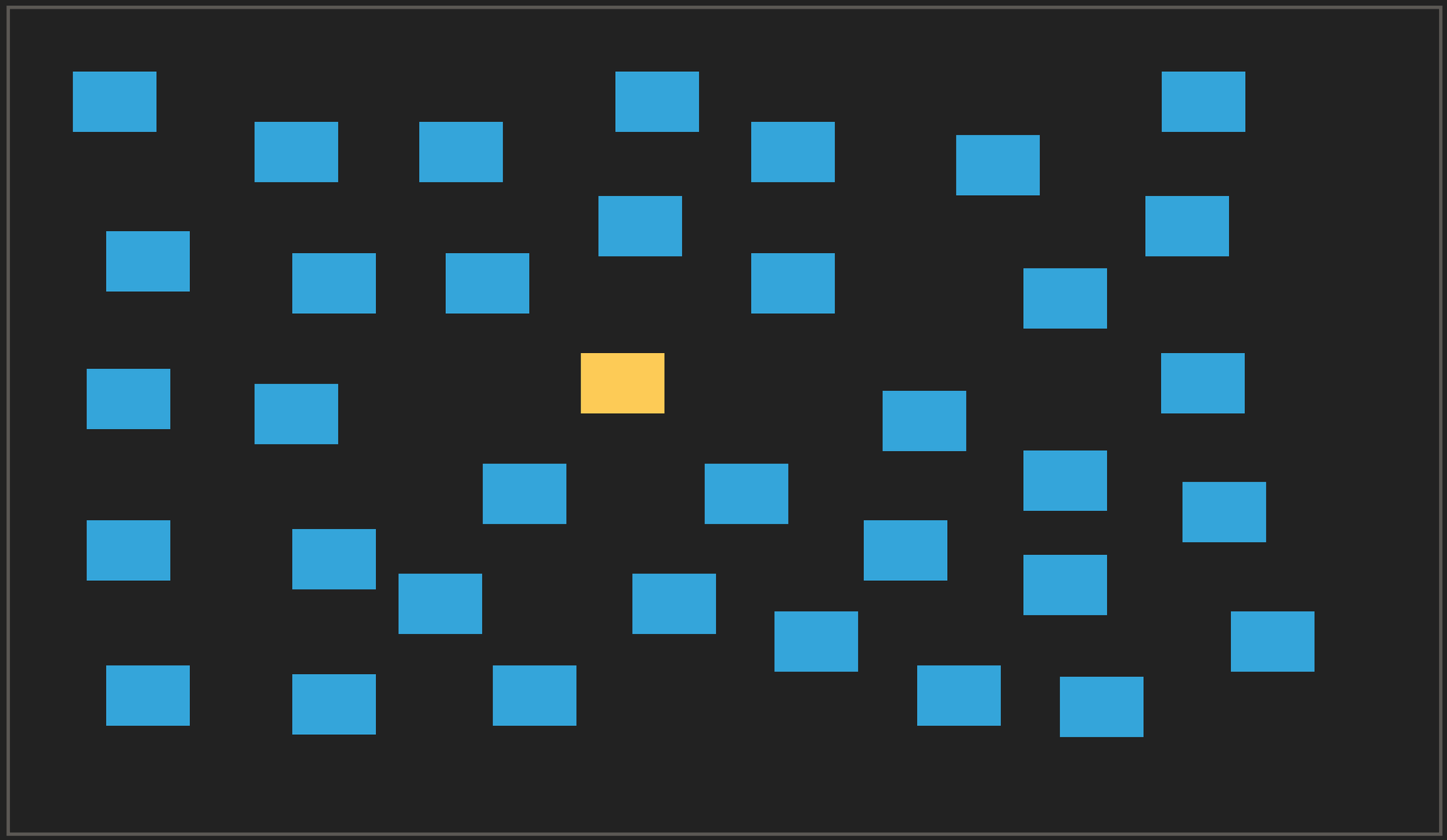
Systems tests

MAINTENANCE COSTS

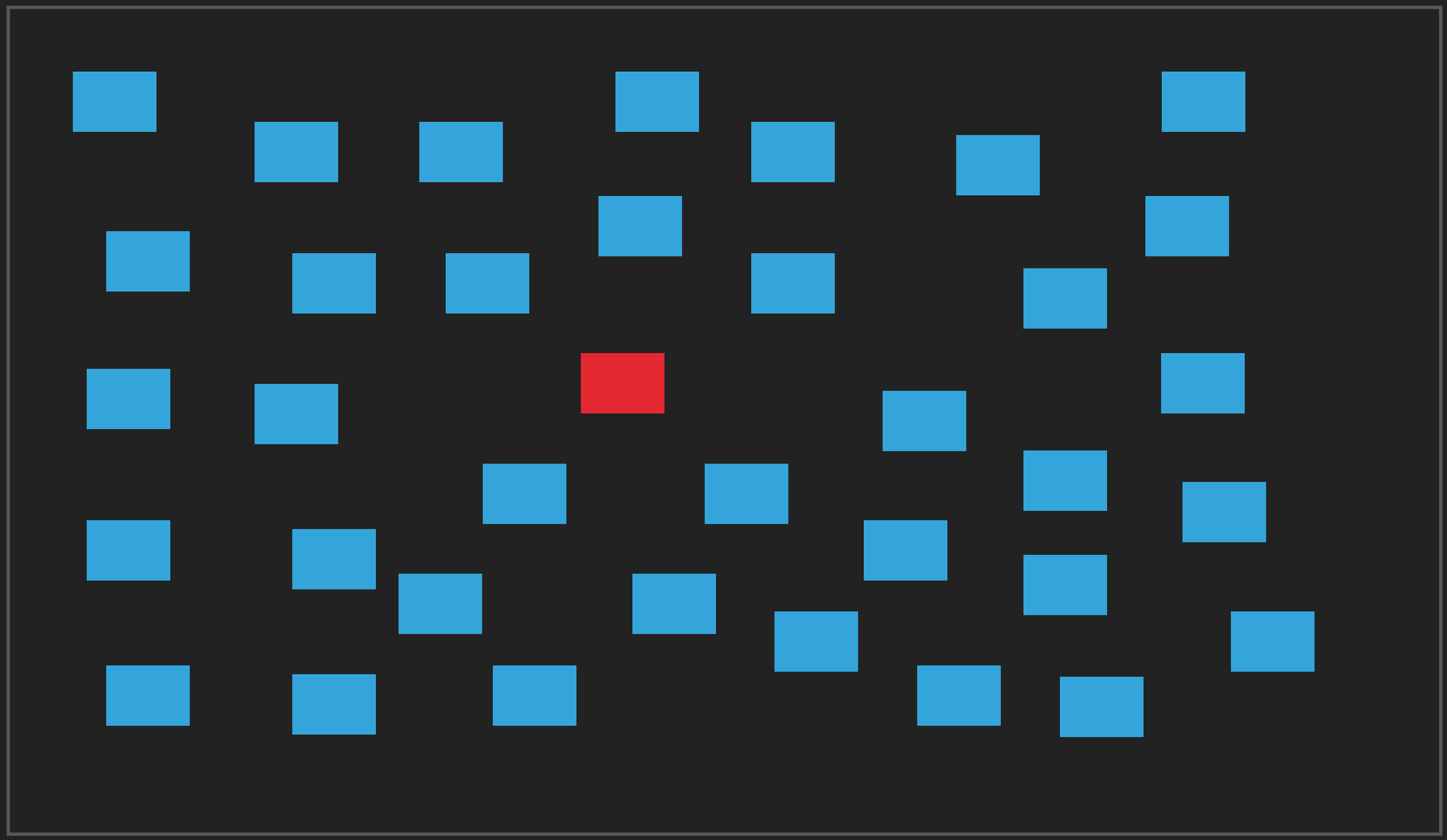
Unit tests

Systems tests

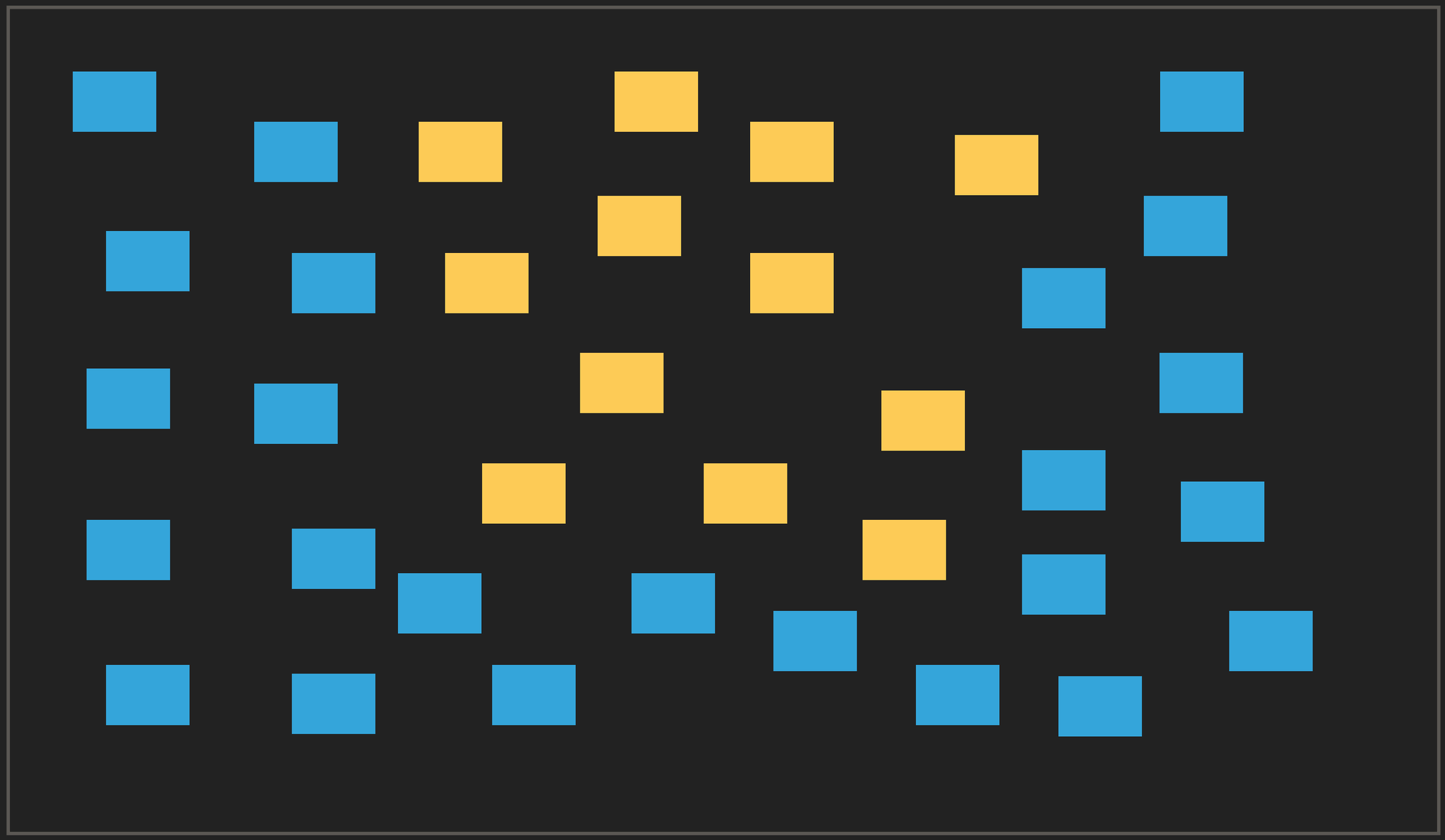
UNIT TEST



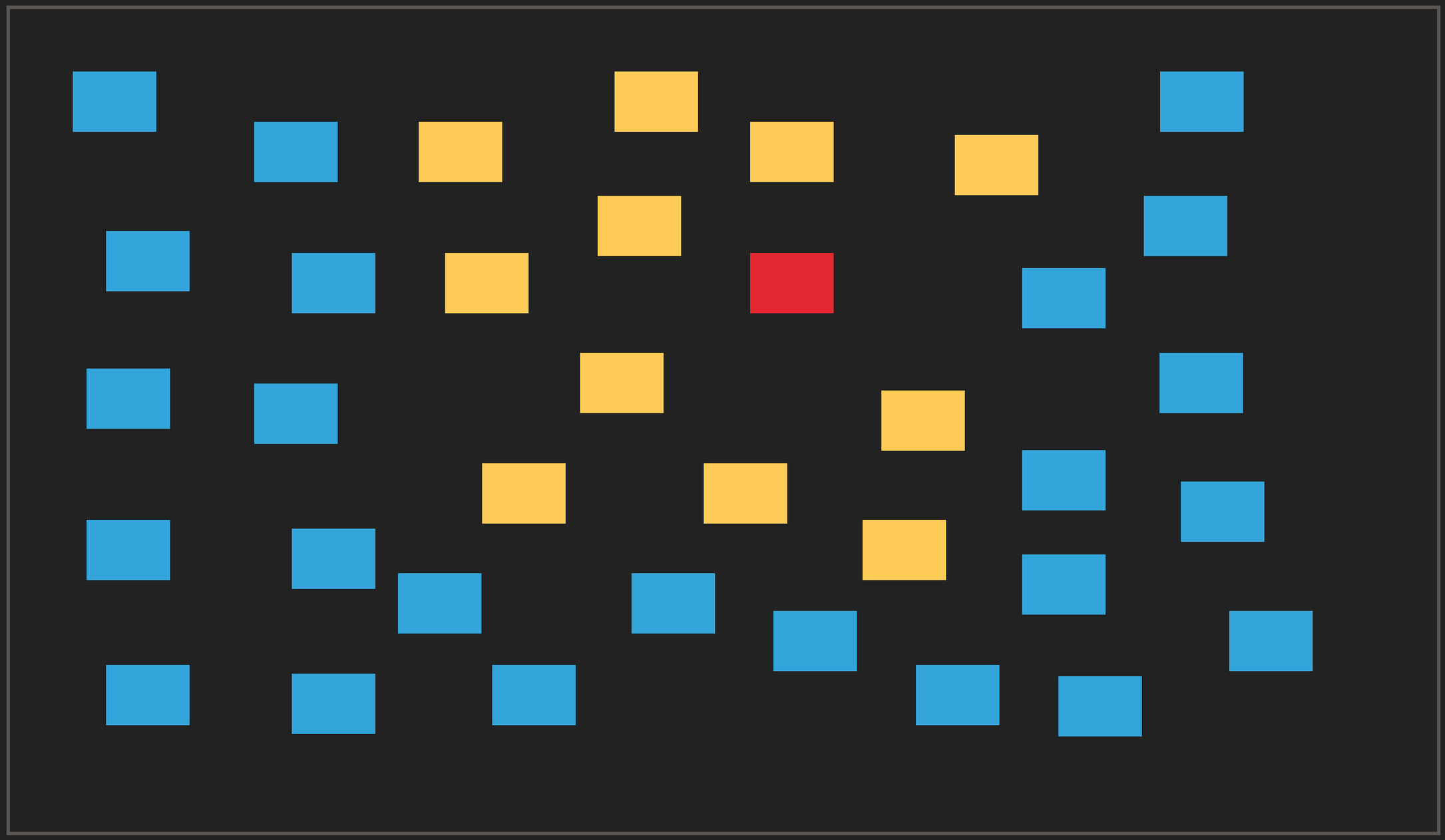
UNIT TEST



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

MAINTENANCE COSTS

Low

High

Low

Unit tests

Systems tests

MAINTENANCE COSTS

Low

High

High

Low

Unit tests

Systems tests

SO FAR NOTHING TOO CONTROVERSIAL

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

EVERYTHING IS COMPROMISE

NOTHING IS BLACK AND WHITE

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

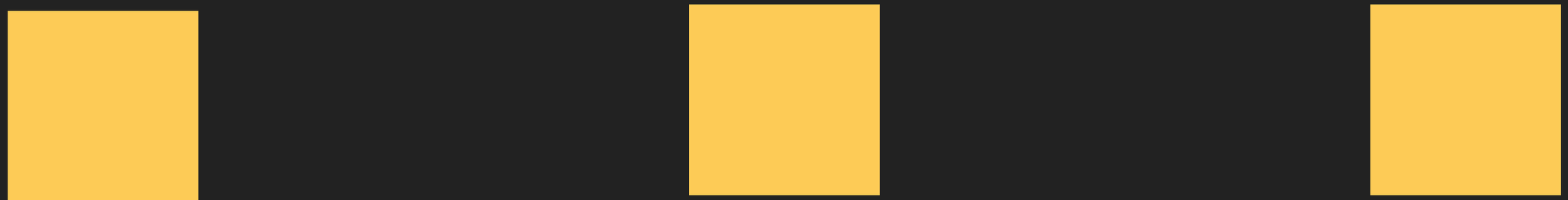
Systems tests

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

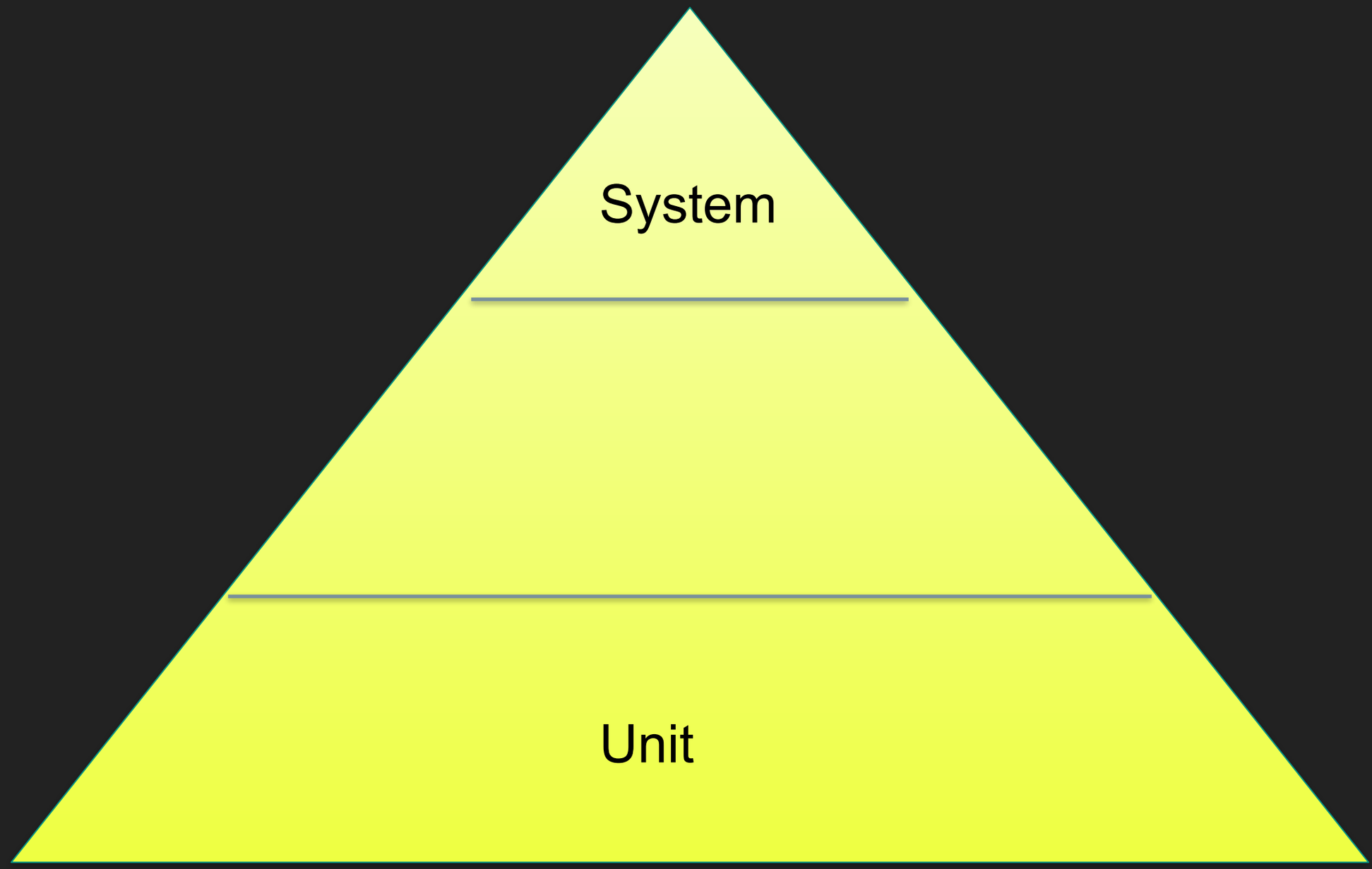
Systems tests

WHERE ALONG THE CONTINUUM SHOULD WE TEST?

Unit tests

Systems tests

TEST PYRAMID



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

- Not achievable
  - Our goals contradict each other
- Nothing is black and white

UNIT TESTS IN MORE DEPTH

UNIT TEST EXAMPLE – SOFTWARE UNDER 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

UNIT TEST EXAMPLE – TEST CASES REQUIRED

- Valid passwords:
  - “Passw0rd”

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

LOOK HOW EASY IT IS TO 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);
    }
}
```

UNIT TEST THIS KIND OF LOGIC

- Unit test sweet spot
- Quicker to test than not test
- Learn how to use data providers for your test framework

FABULOUS FIVE

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

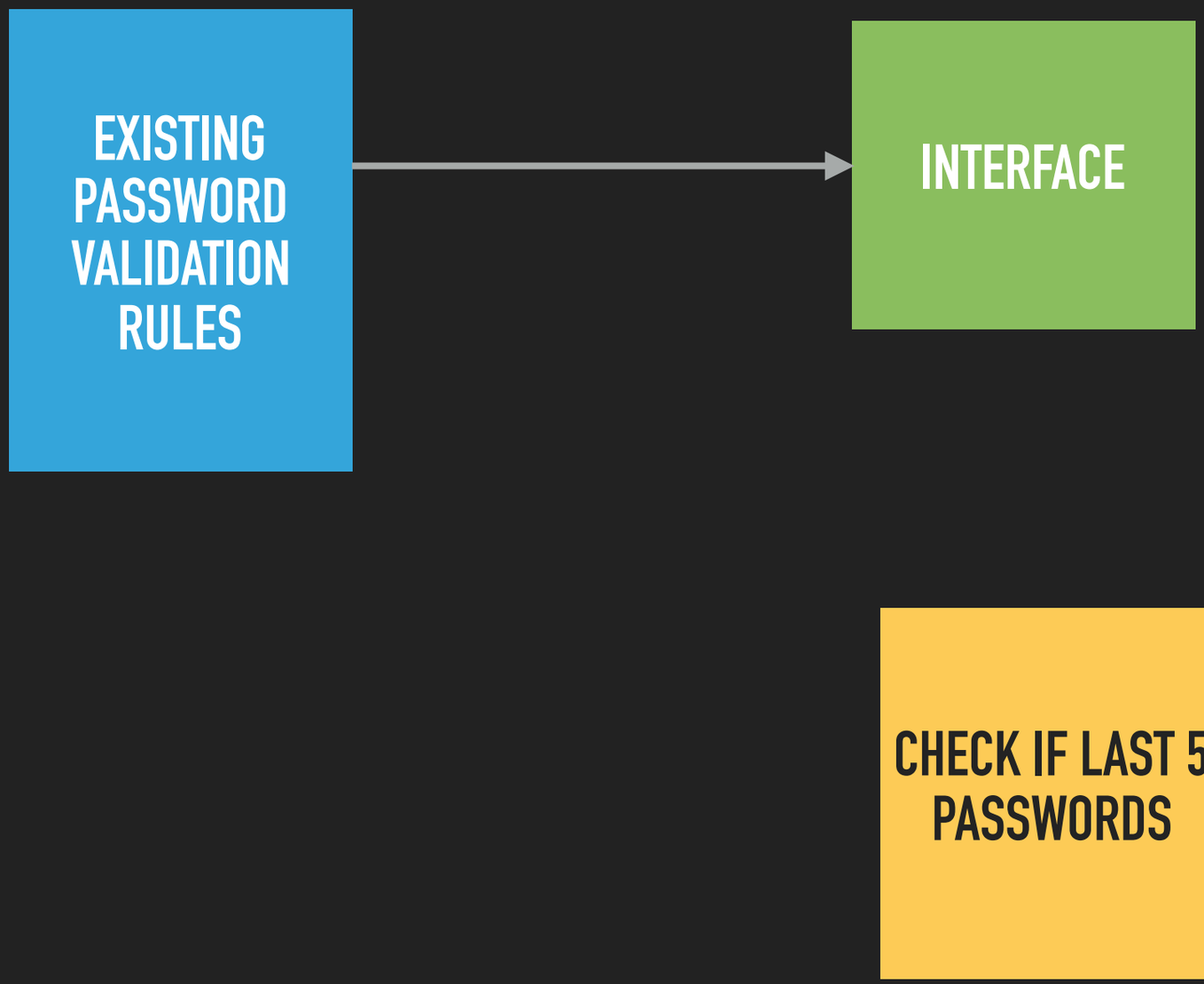
ARCHITECTURE

EXISTING PASSWORD VALIDATION RULES

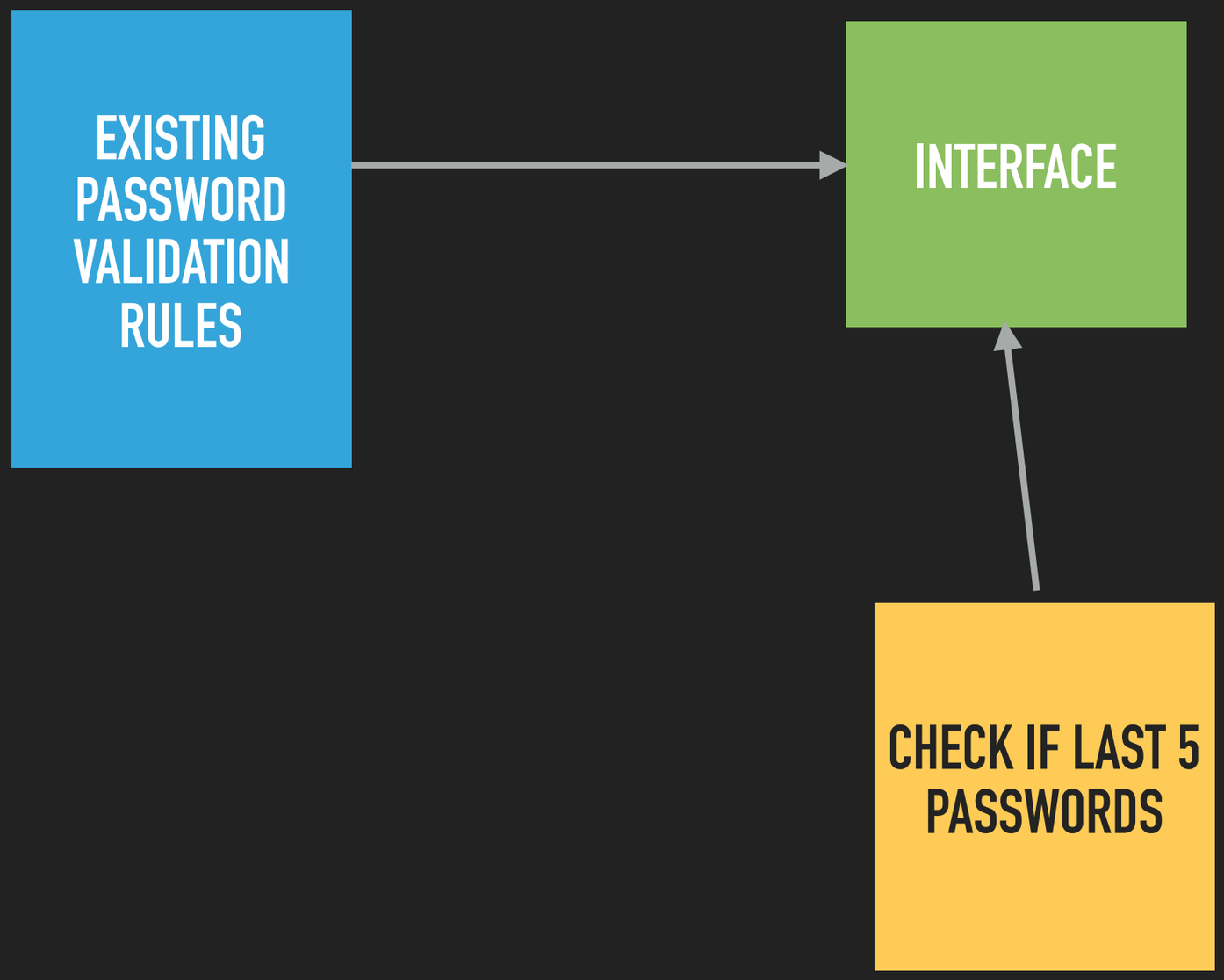
INTERFACE

CHECK IF LAST 5 PASSWORDS

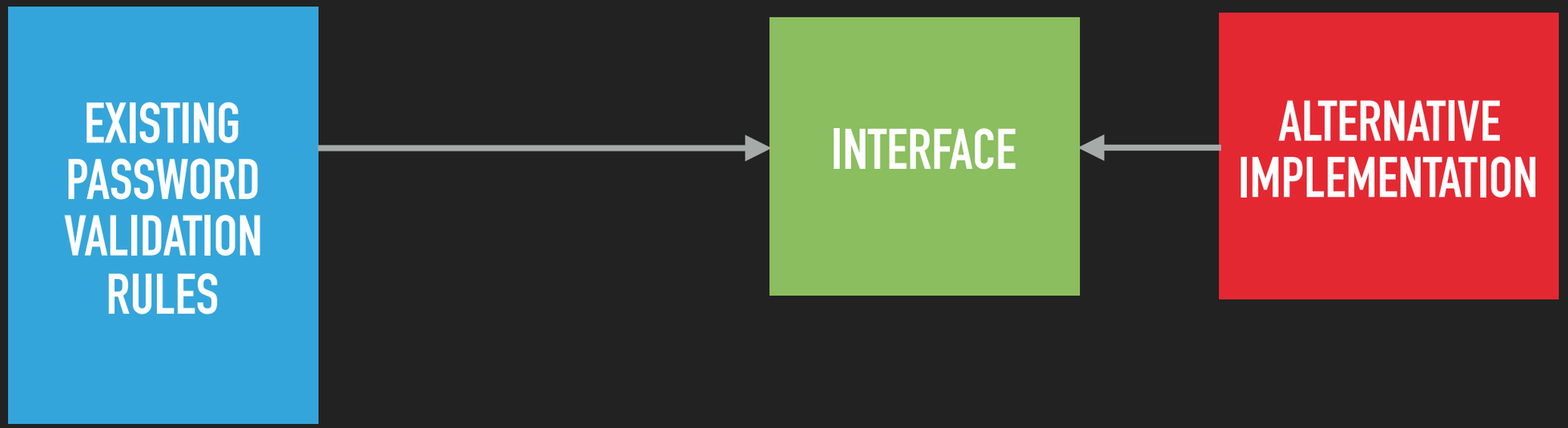
ARCHITECTURE



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);

}
```

OPTIONS WITH DEPENDENCIES

OPTIONS WITH DEPENDENCIES

- Real thing

OPTIONS WITH DEPENDENCIES

- Real thing
- Test double

OPTIONS WITH DEPENDENCIES

- Real thing
- Test double
- Stub

OPTIONS WITH DEPENDENCIES

- Real thing
- Test double
  - Stub
  - Mock

OPTIONS WITH DEPENDENCIES

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

PASSWORD VALIDATOR TEST REVISITED

PASSWORD VALIDATOR TEST REVISITED

- Update existing tests to account for:
  - Any calls to RecentPasswordChecker

PASSWORD VALIDATOR TEST REVISITED

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

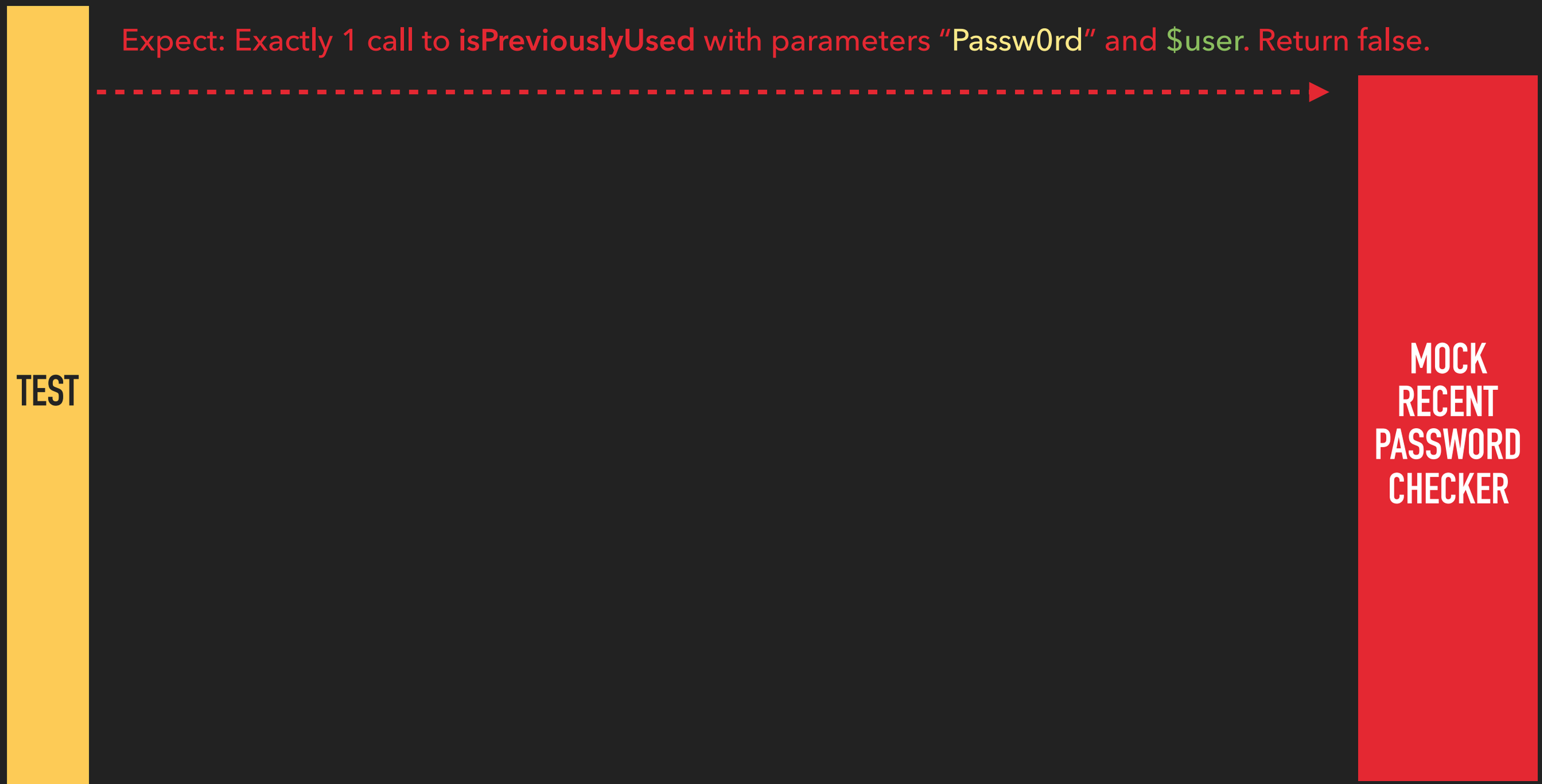
NEW TEST: VALID PASSWORD, NOT RECENTLY USED

NEW TEST: VALID PASSWORD, NOT RECENTLY USED

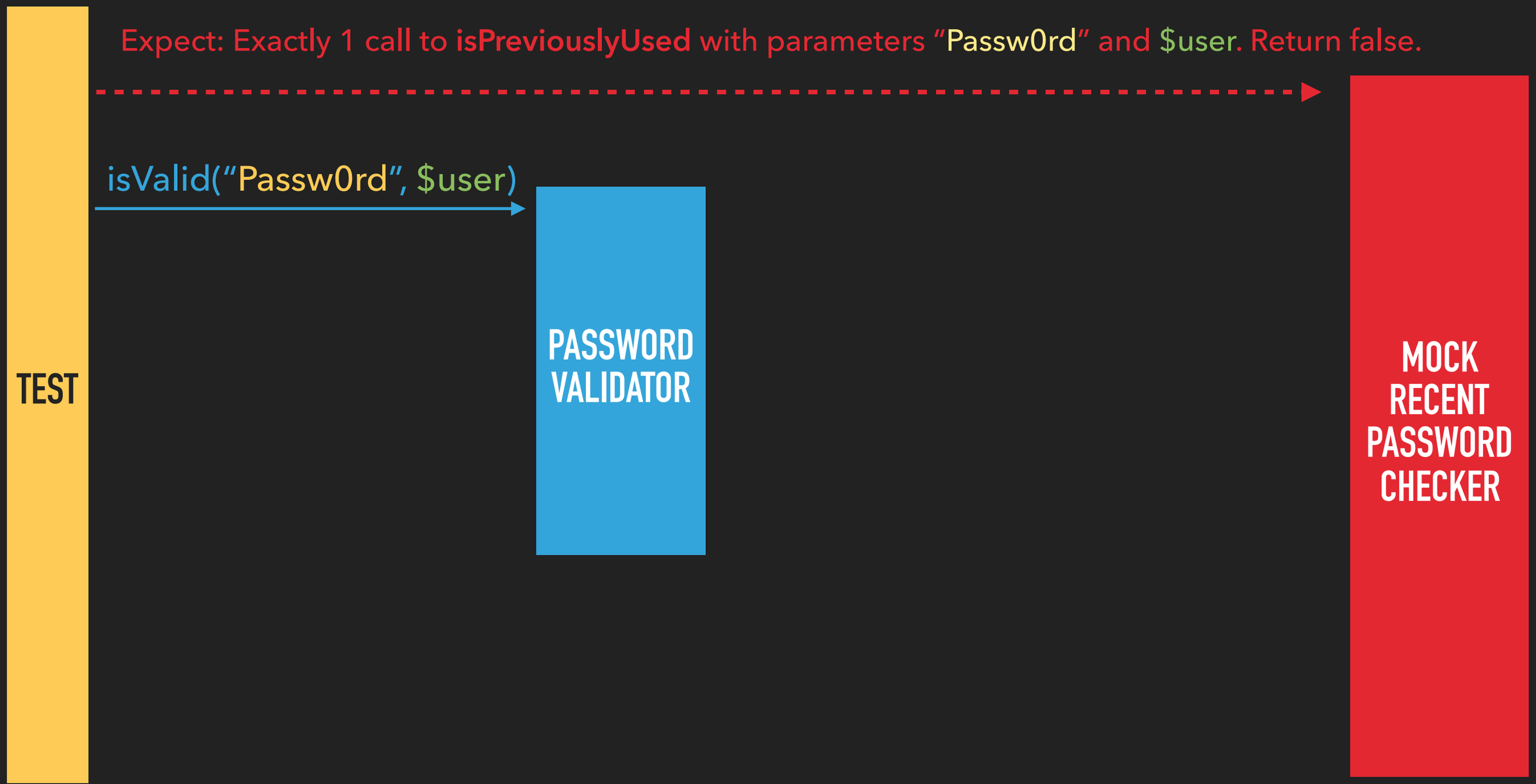
TEST

NEW TEST: VALID PASSWORD, NOT RECENTLY USED

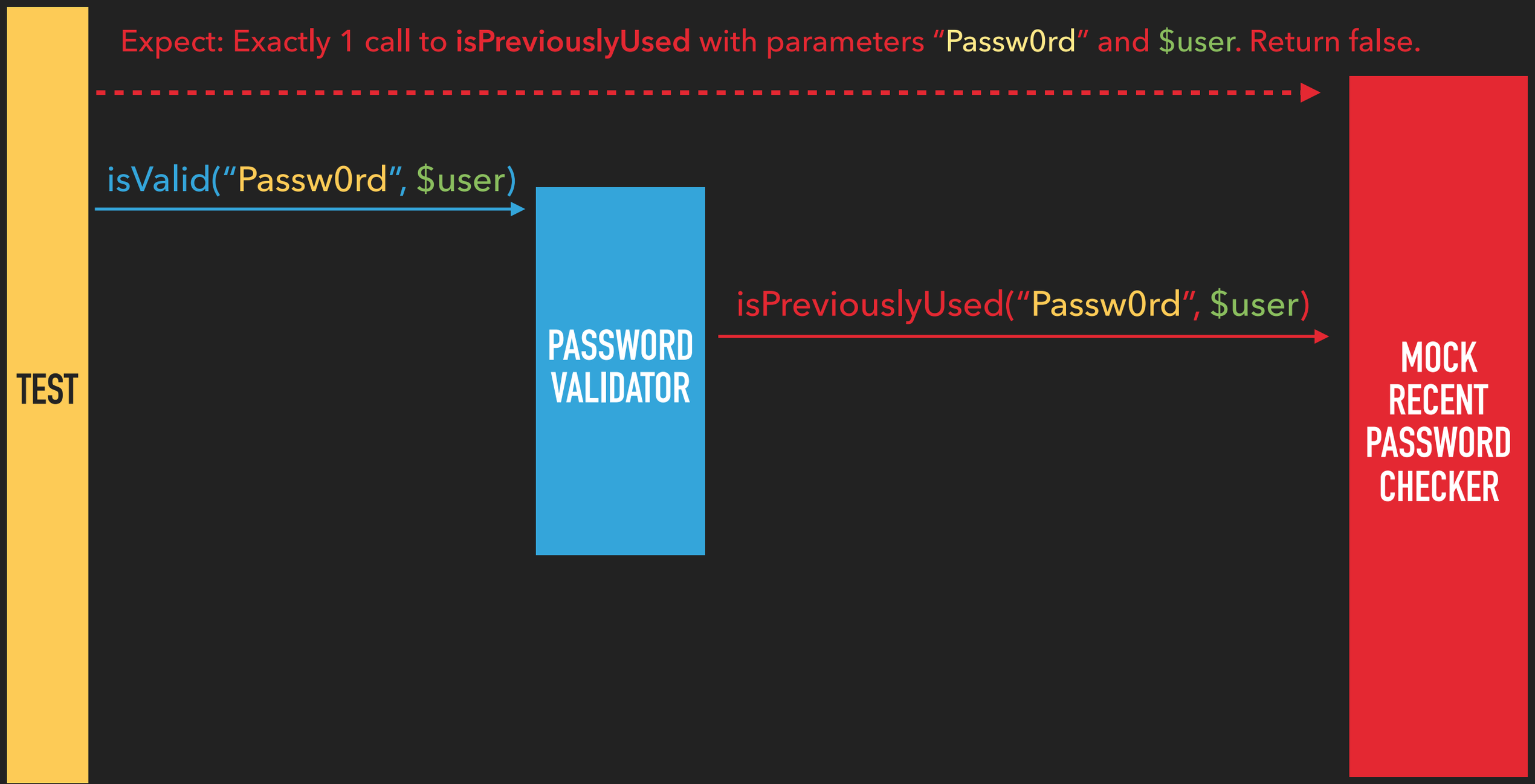
Expect: Exactly 1 call to **isPreviouslyUsed** with parameters "Passw0rd" and $user. Return false.



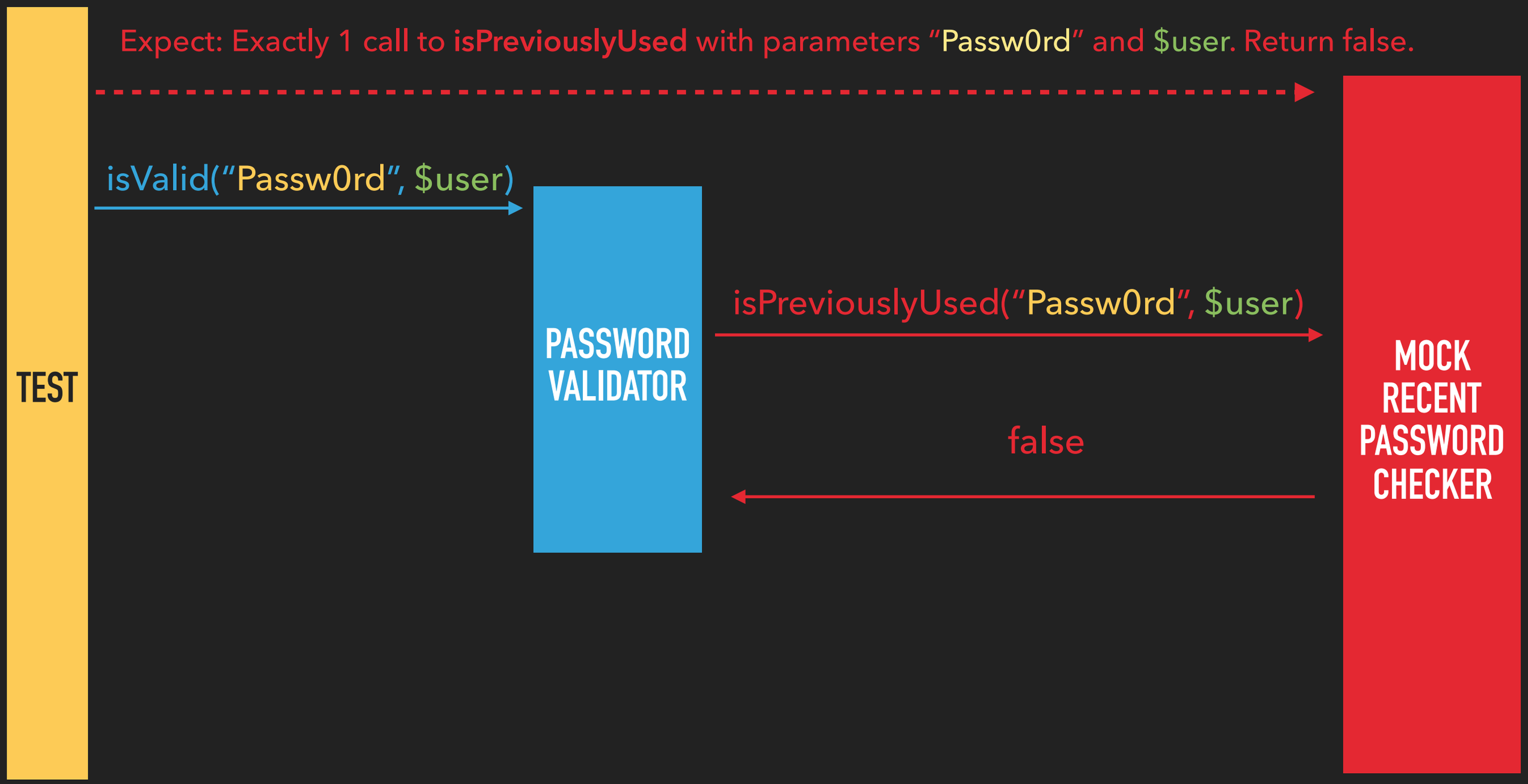
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



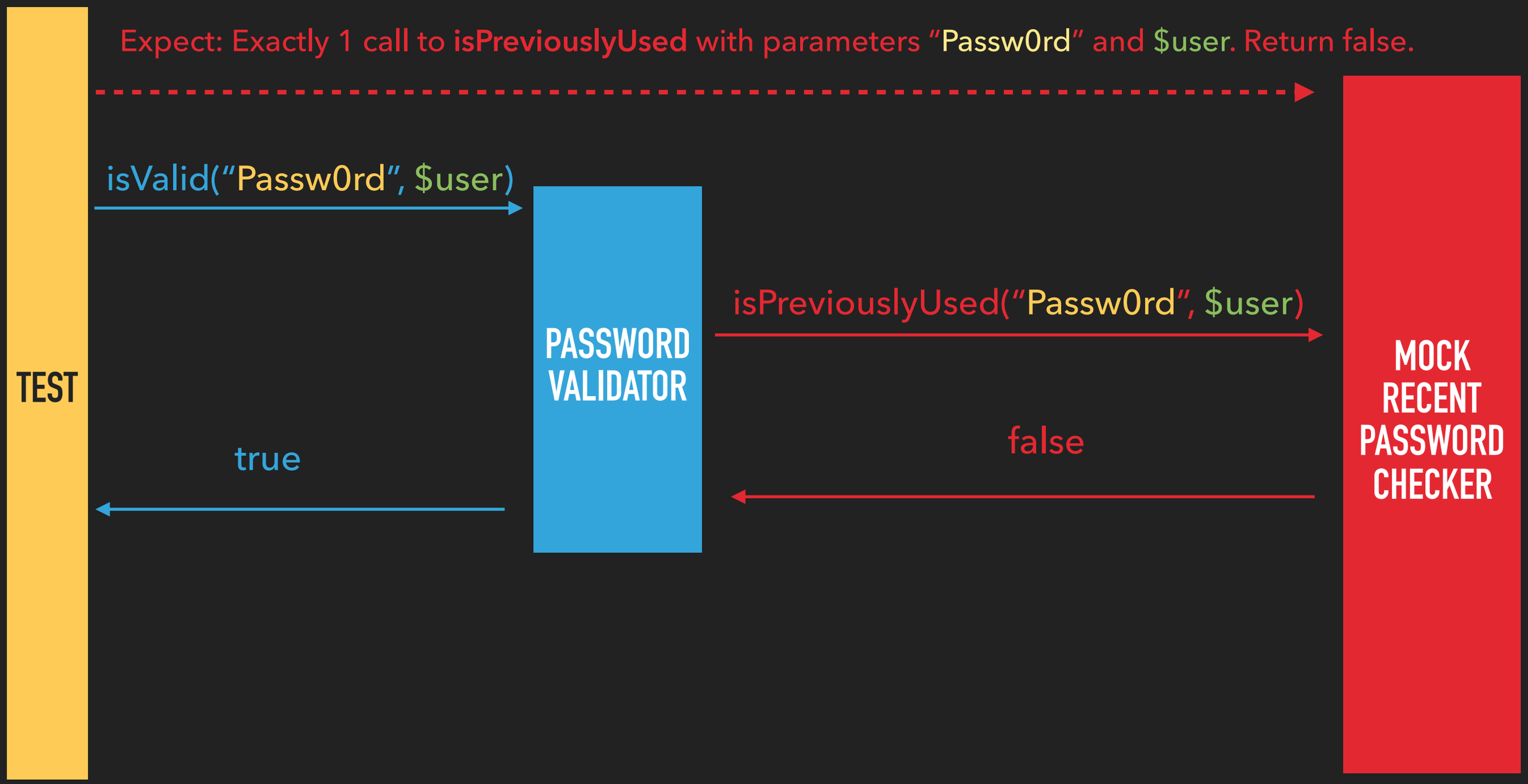
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



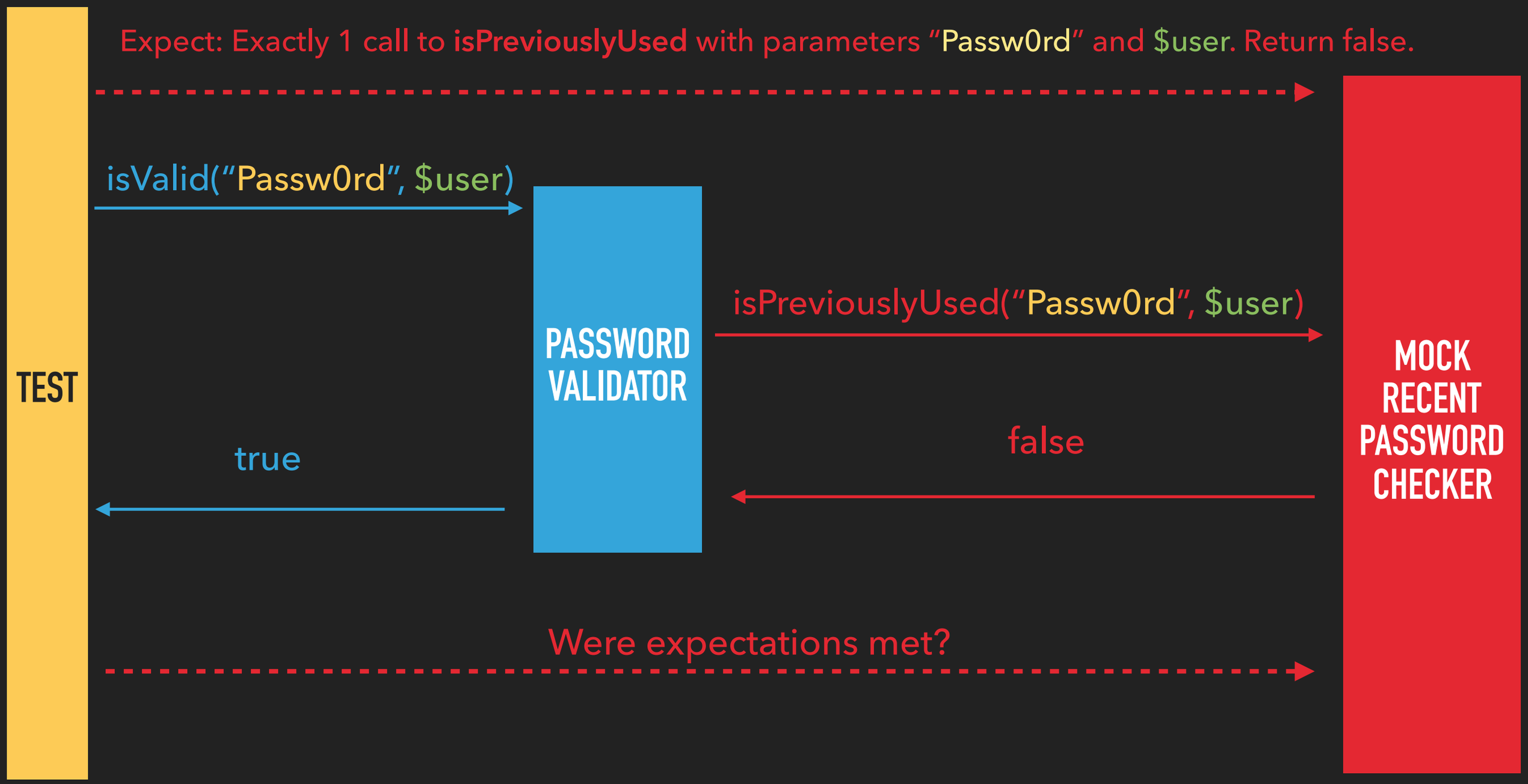
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



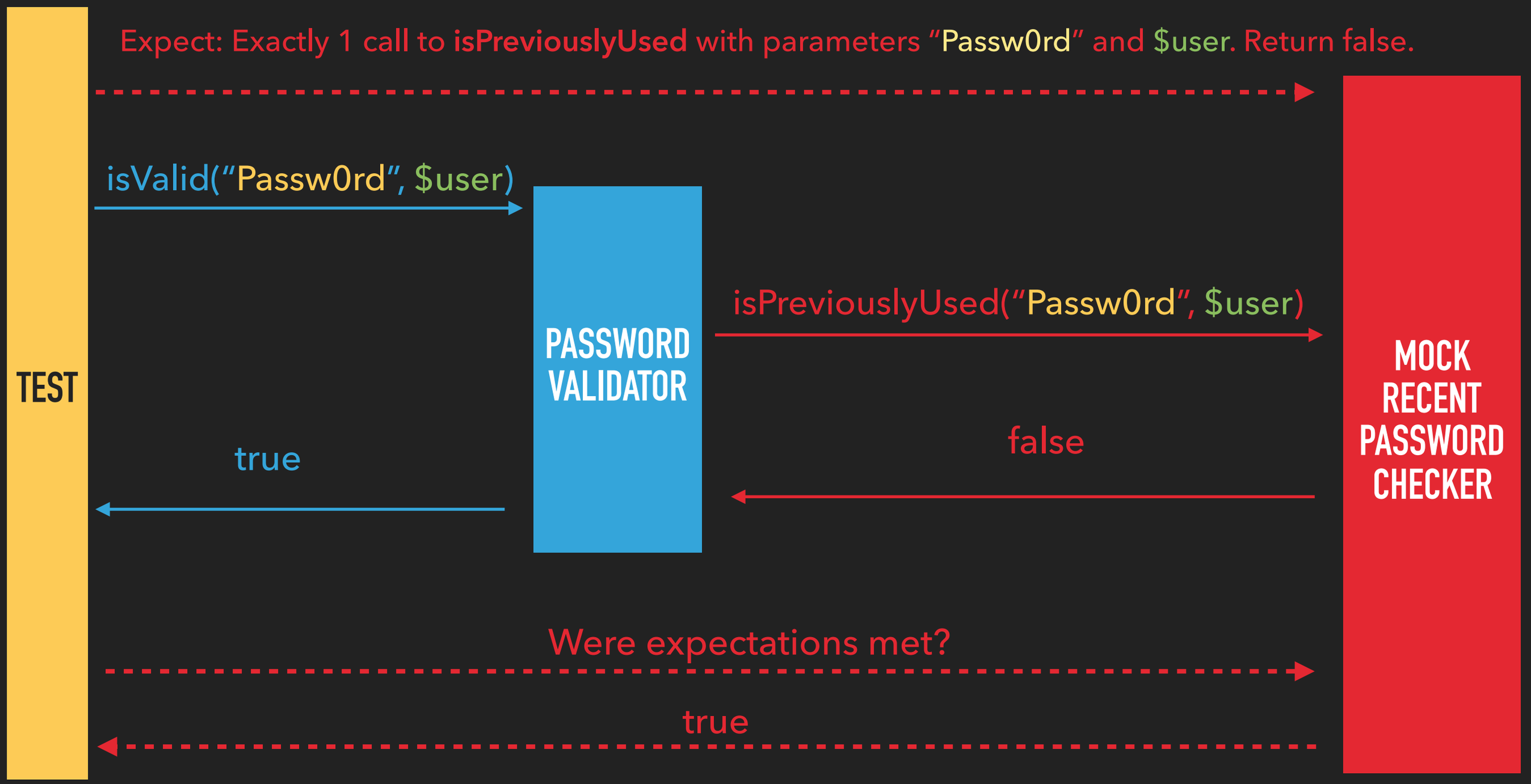
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



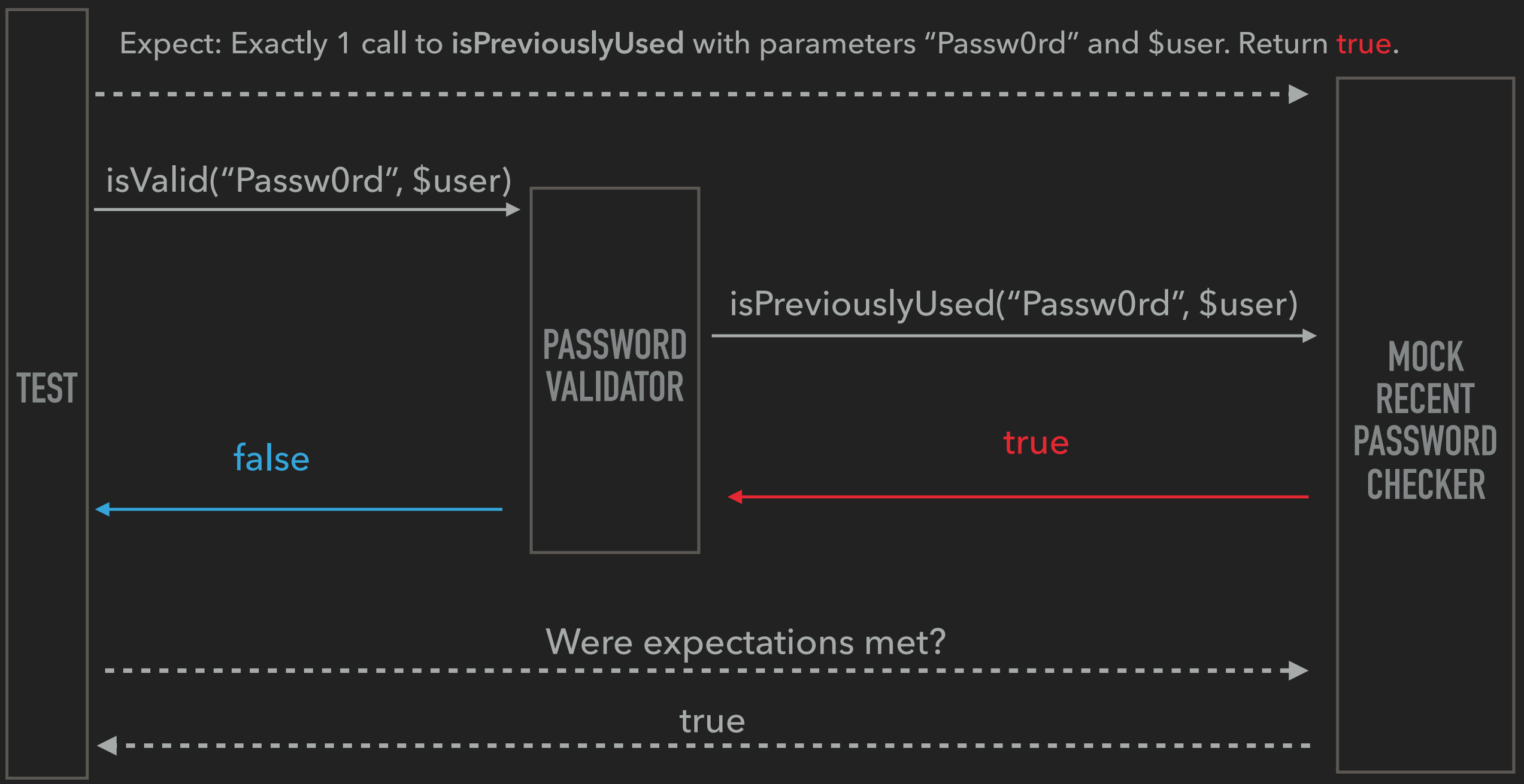
NEW TEST: VALID PASSWORD, NOT RECENTLY USED



NEW TEST: VALID PASSWORD, NOT RECENTLY USED



NEW TEST: VALID PASSWORD, BUT RECENTLY USED



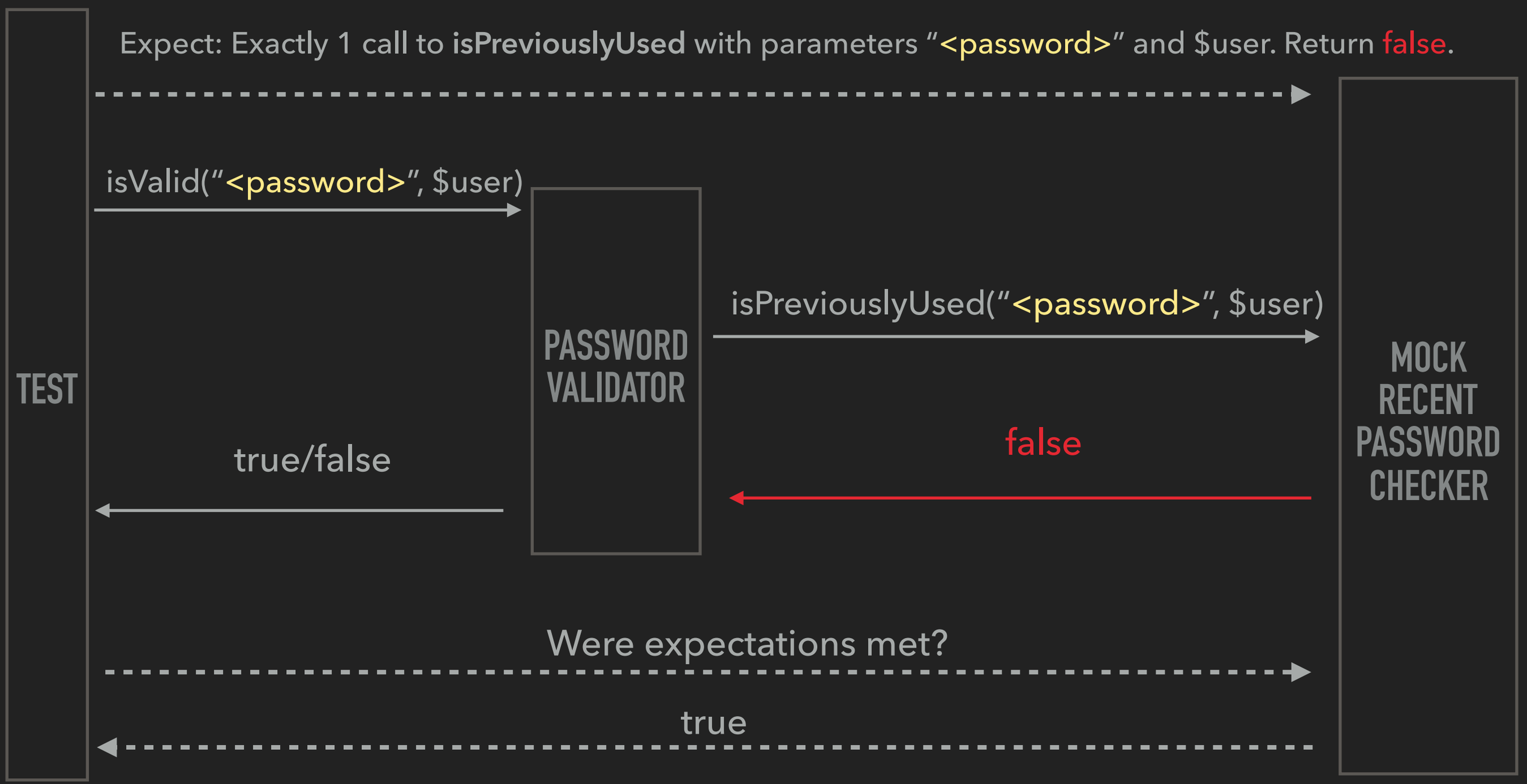
THESE EXTRA 2 TESTS ARE THE AWKWARD DUO

EXISTING TESTS (FABULOUS FIVE)

EXISTING TESTS (FABULOUS 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

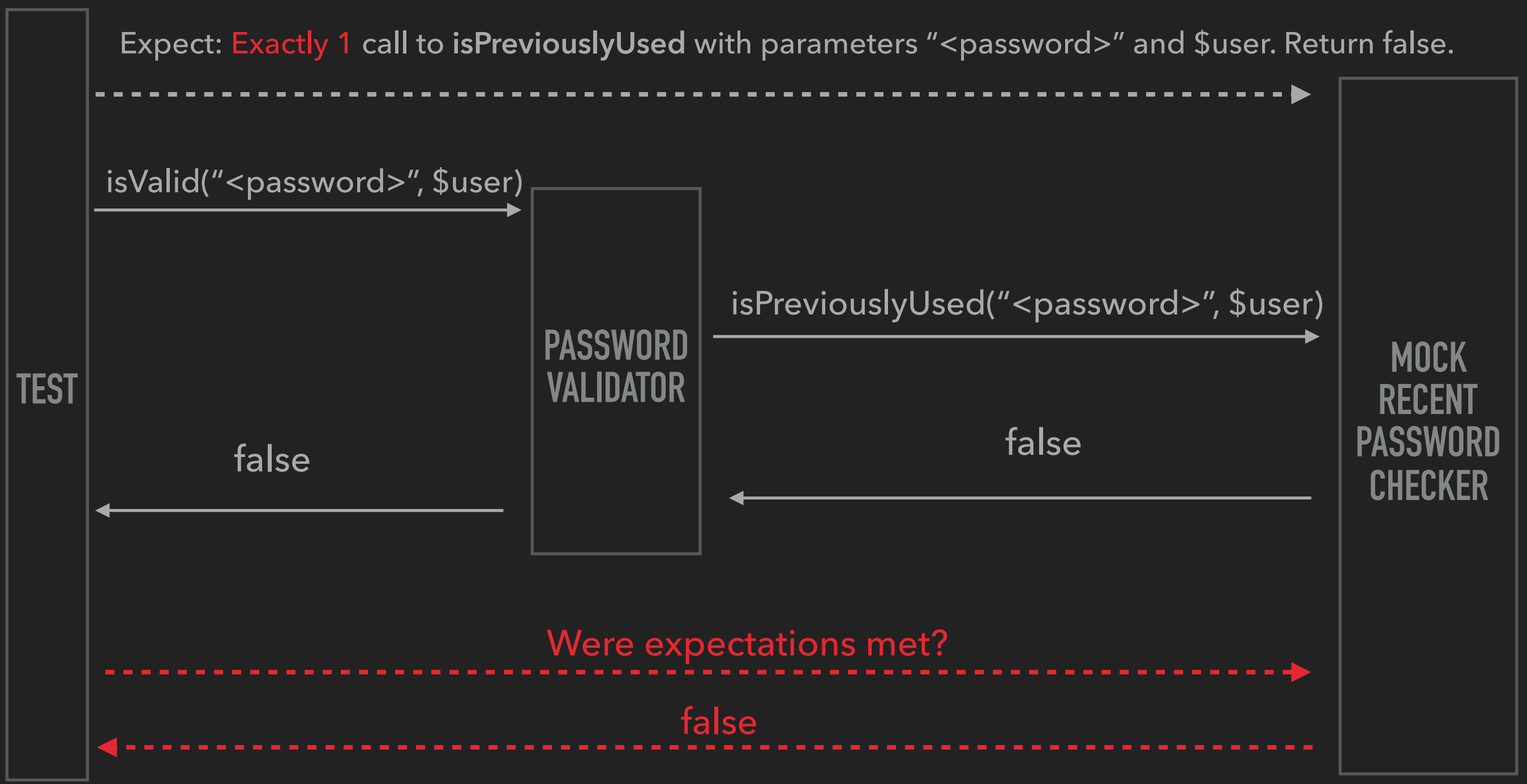
```
class PasswordValidator
{
  public function isValid(string $password, User $user) : bool
  {
     if ($this->recentPasswordChecker->isRecentPassword(
              $password, $user)) {
       return false;
     }

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

    … remaining checks …

    return true;
}
```

EXISTING TESTS – 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;
```

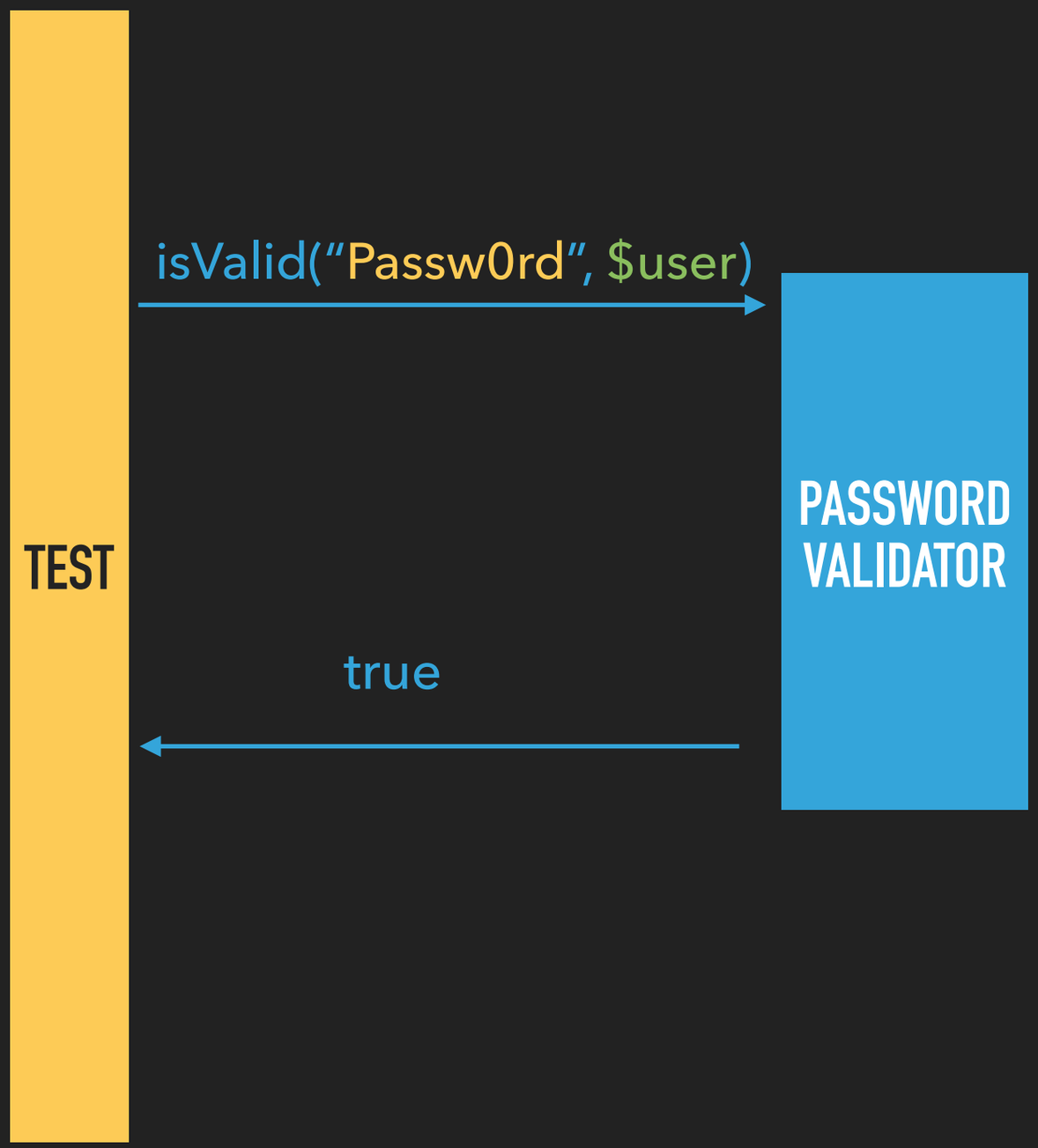
```
   … remaining checks …

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

 return true;
}
```

EXISTING TESTS: AFTER REFACTOR

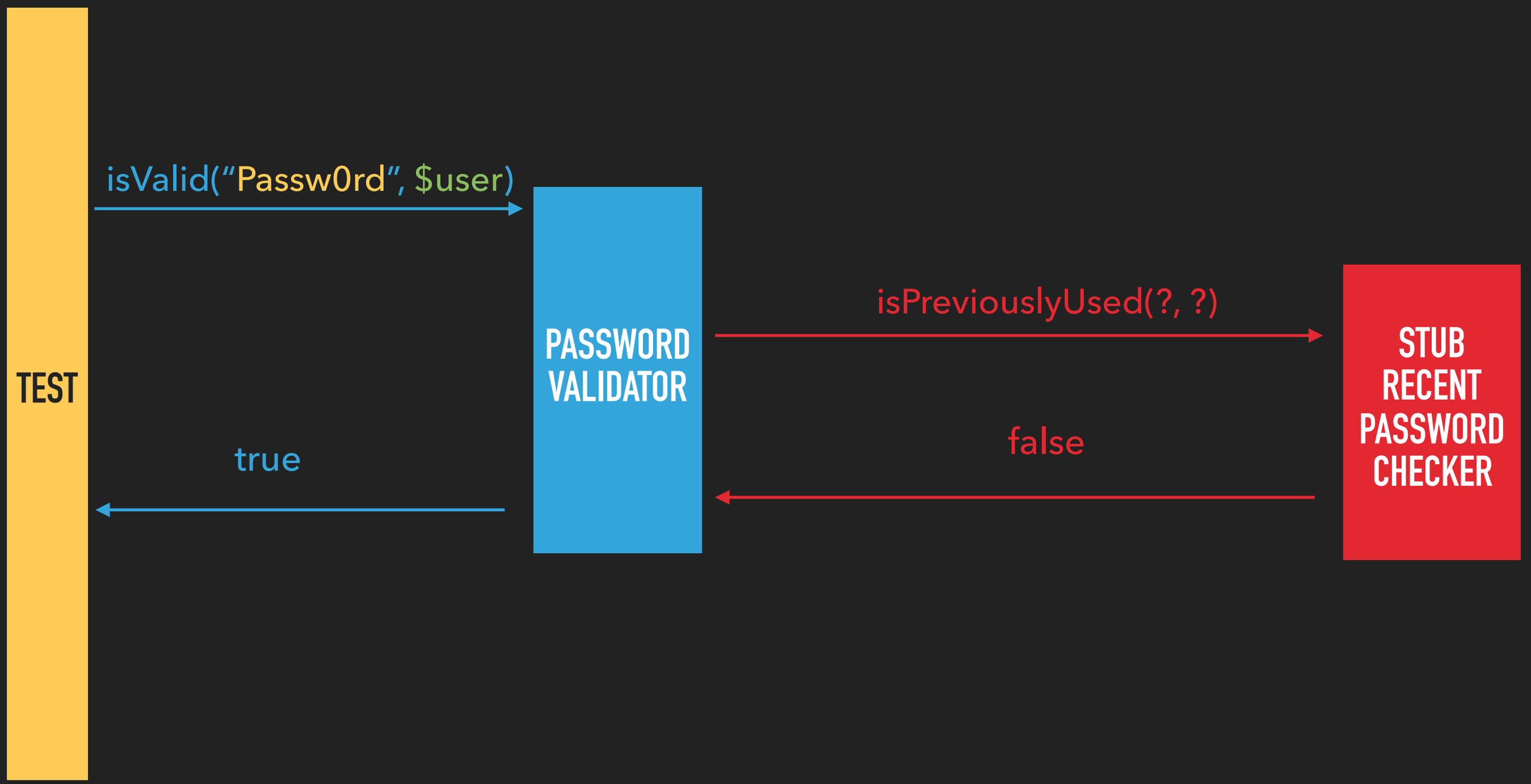


WE'VE REFACTORED CODE AND THE TESTS HAVE BROKEN. NOT GOOD!

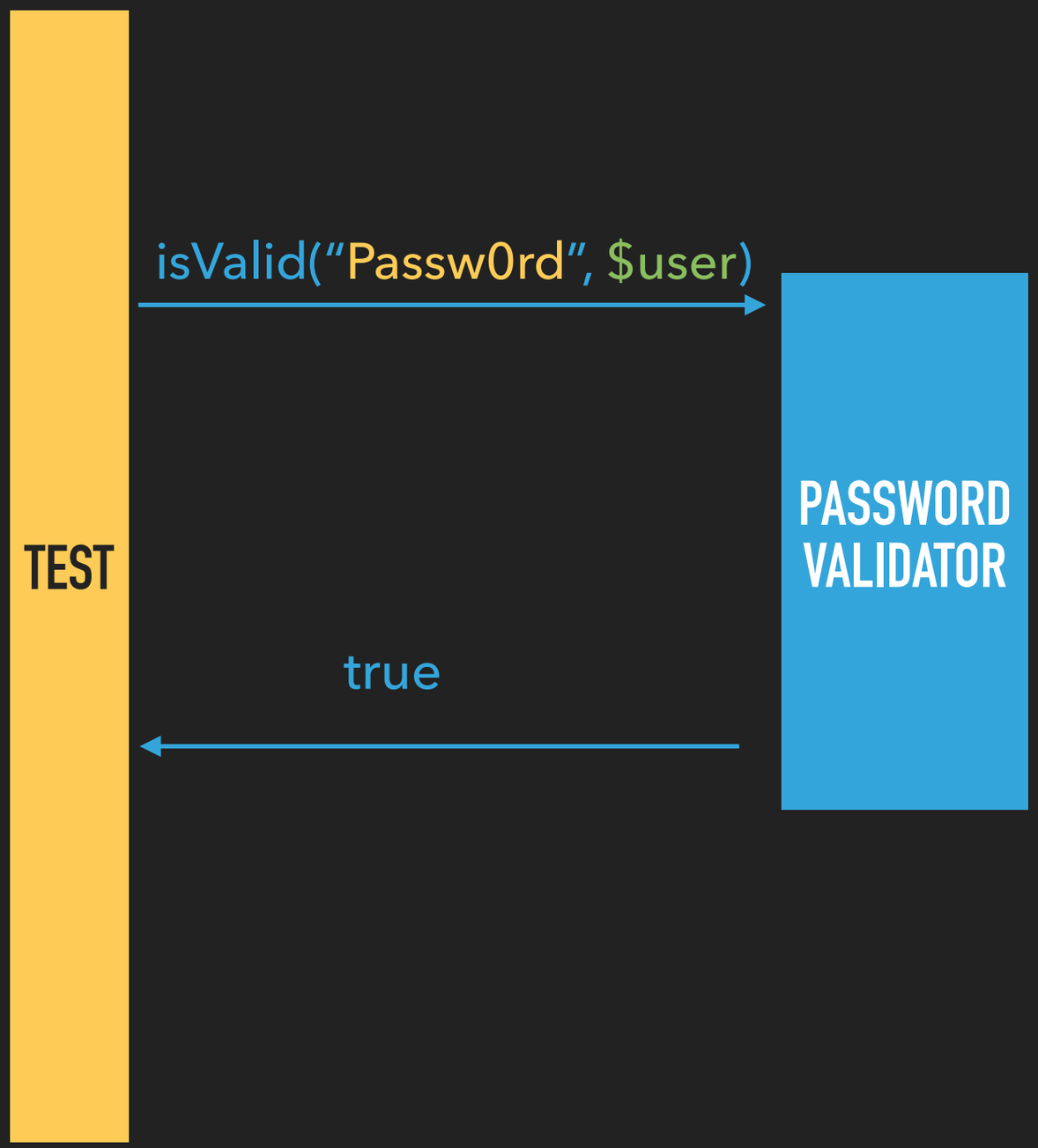
USE A STUB



USE A STUB


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

USE A STUB



TAKE AWAY

USE STUBS UNLESS YOU REALLY NEED MOCKS

- Mocks increase coupling between tests and code
  - Only use them when you really need to
    - Test harder to write
    - Reduces ability to refactor

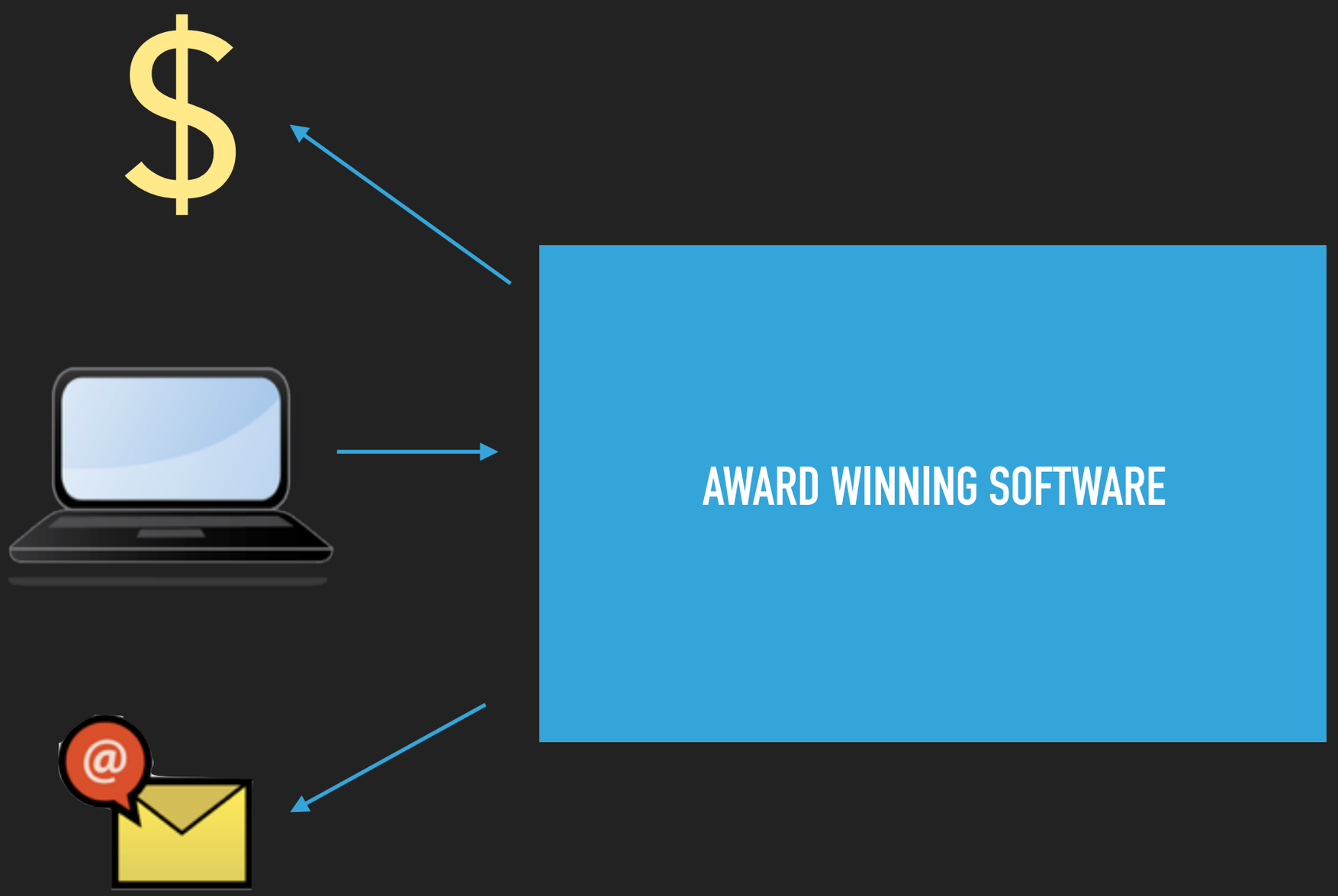
TAKE AWAY

REDUCE COUPLING BETWEEN YOUR TESTS AND THE IMPLEMENTATION OF THE SOFTWARE UNDER TEST

- Unnecessary coupling increases maintenance cost
  - makes test more fragile
  - reduces ability to refactor

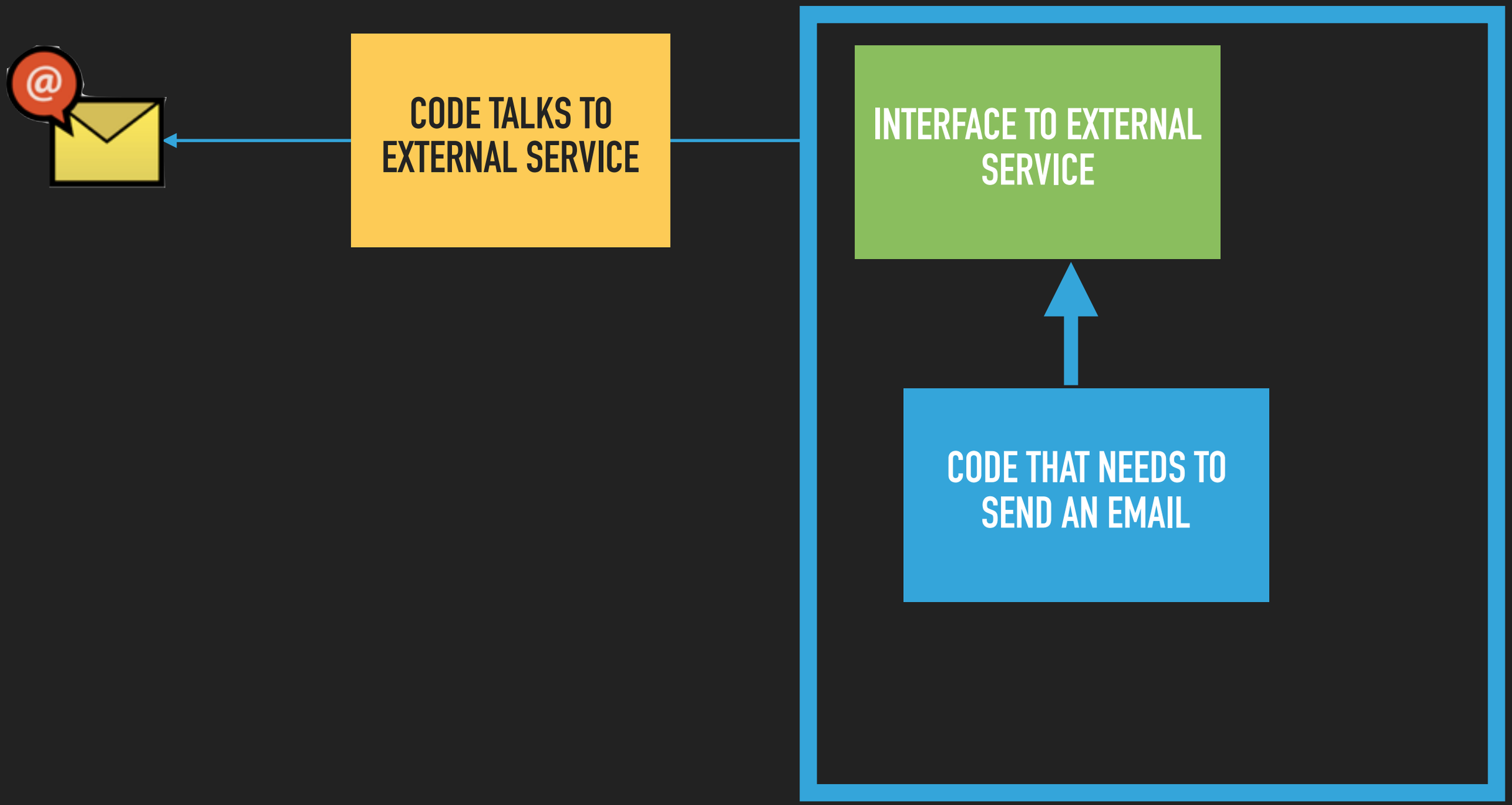
BIGGER TESTS

ARCHITECTURE

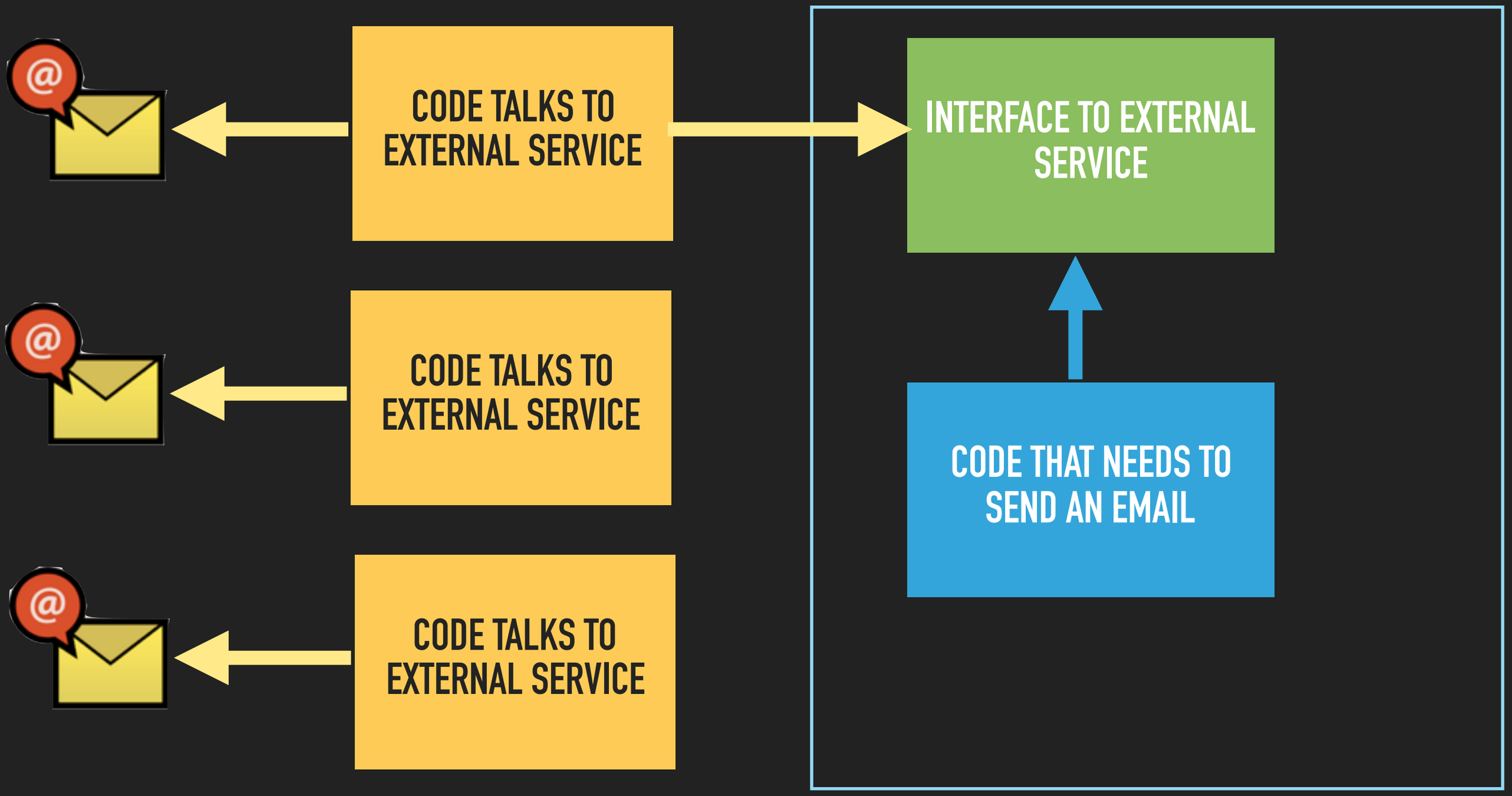


ARCHITECTURE

ARCHITECTURE



ARCHITECTURE



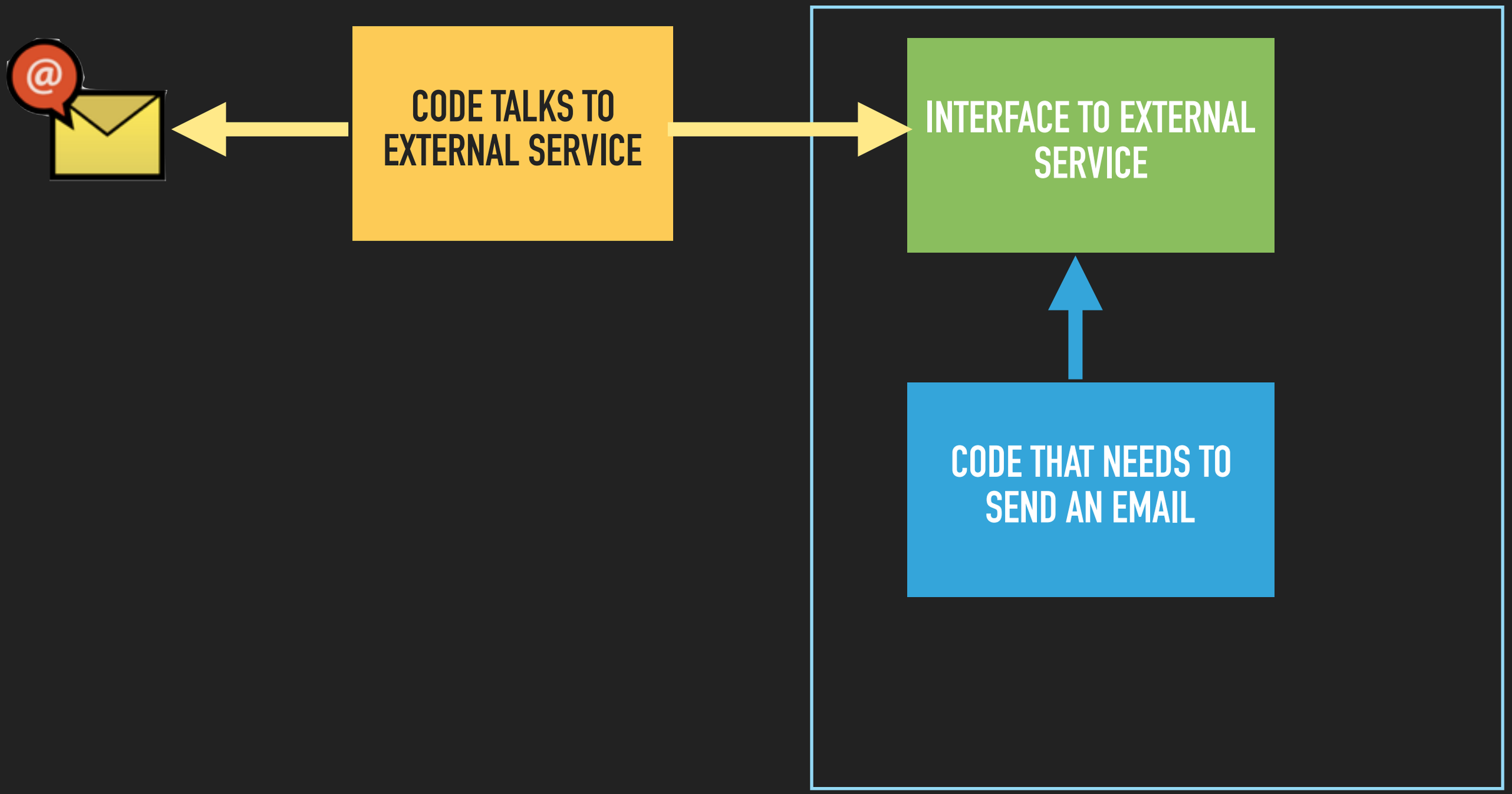
EMAIL GATEWAY INTERFACE

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

}
```

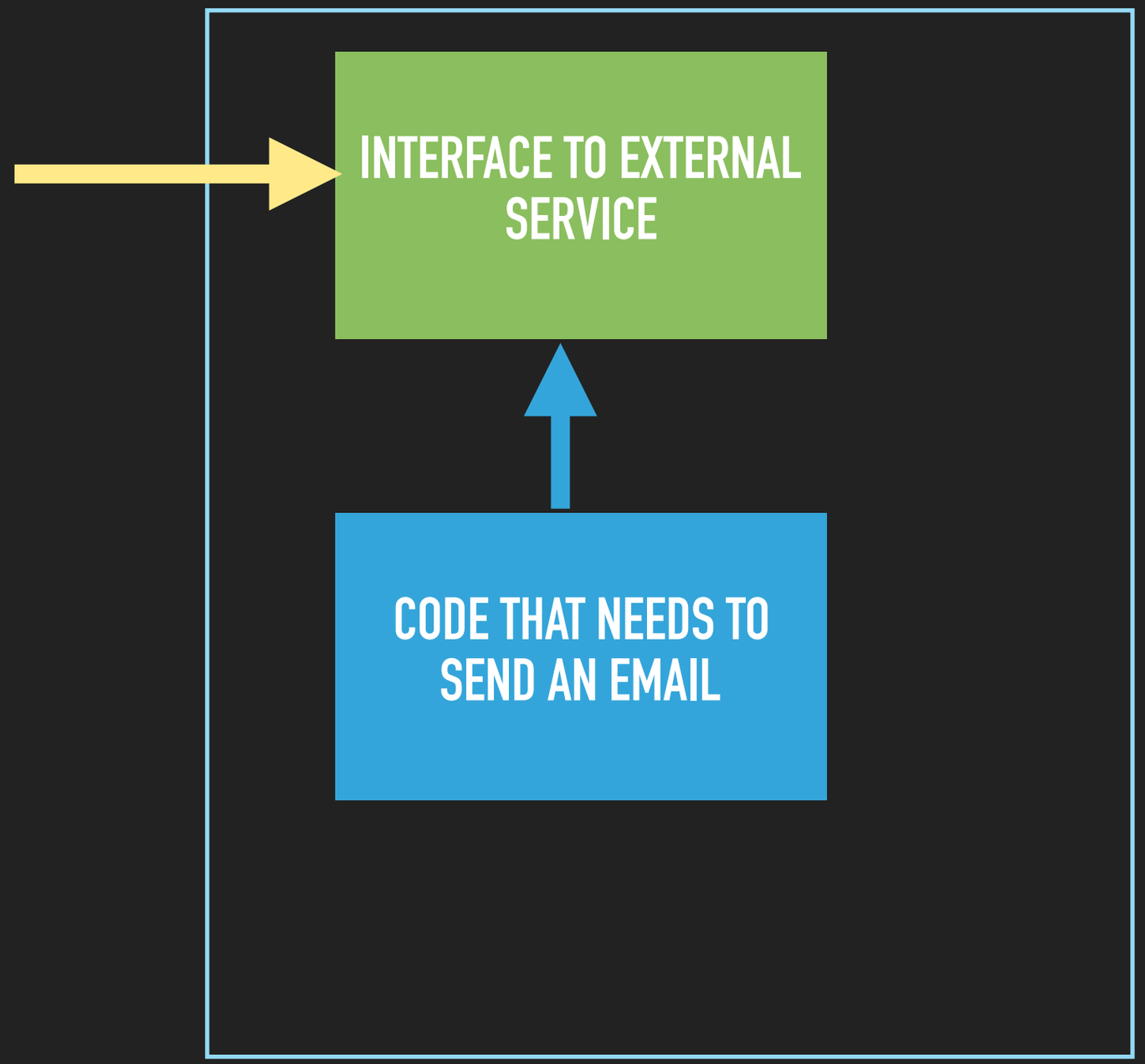
EMAIL MESSAGE OBJECT

- To
- From
- CC
- Subject
- Template name (e.g. REGISTER_USER, PASSWORD_RESET)
- Data

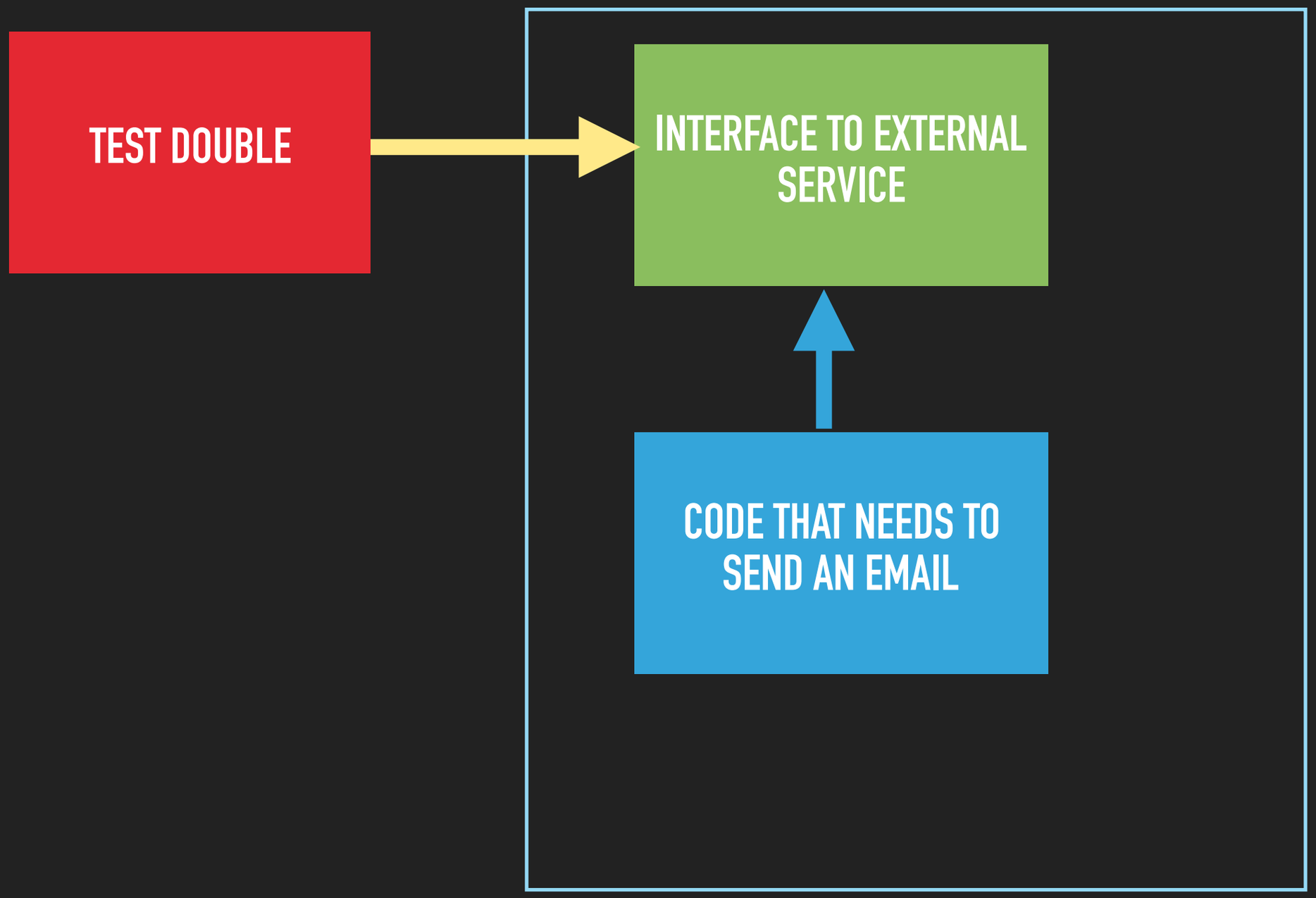
ARCHITECTURE



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 …
  }
}
```

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 …
  }
}
```

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 …
  }
}
```

FAKE EMAIL GATEWAY

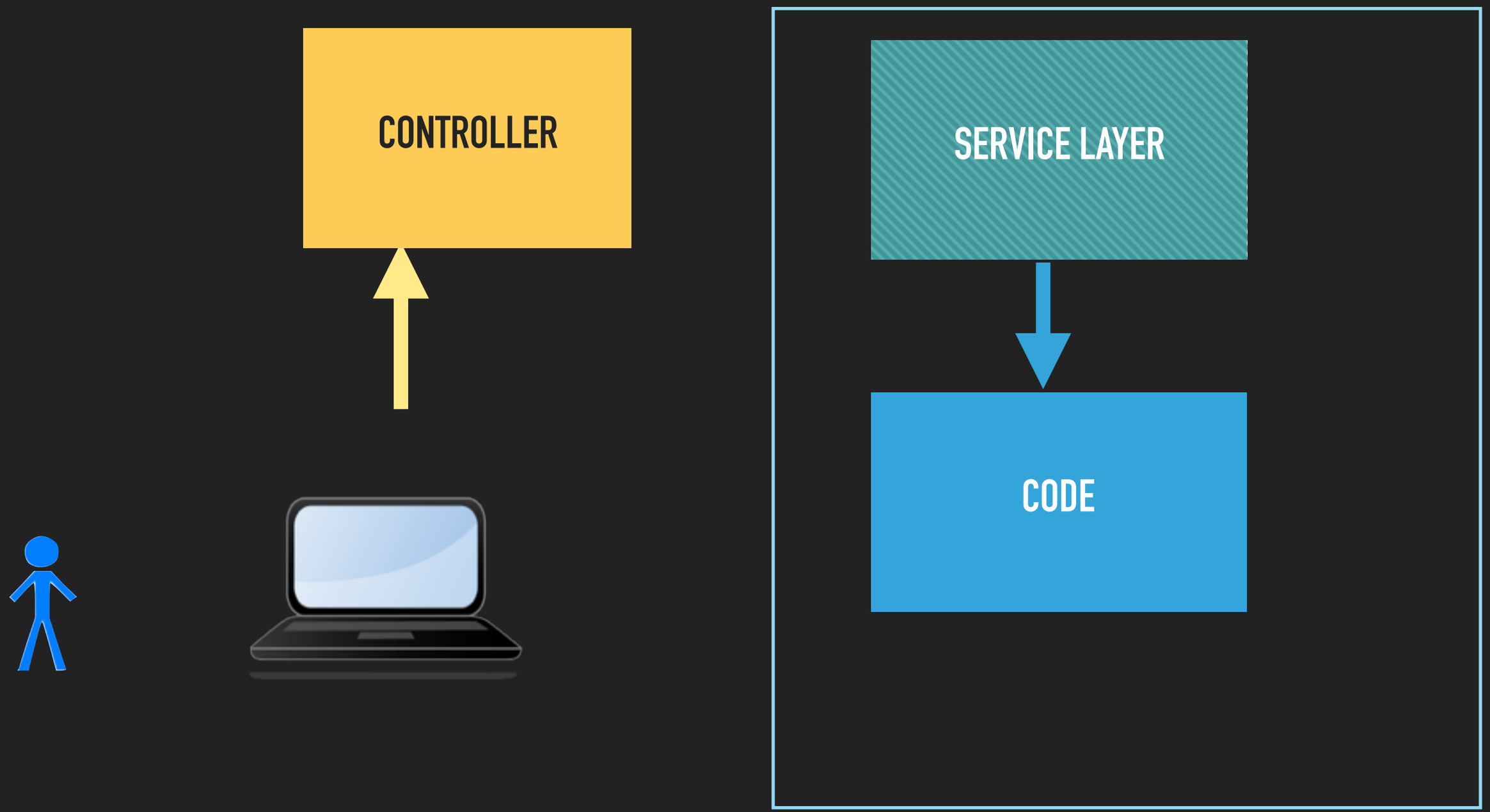
```
class FakeEmailGateway implements EmailGatewayInterface
{

  private $emailMessages = [];

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

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

ARCHITECTURE

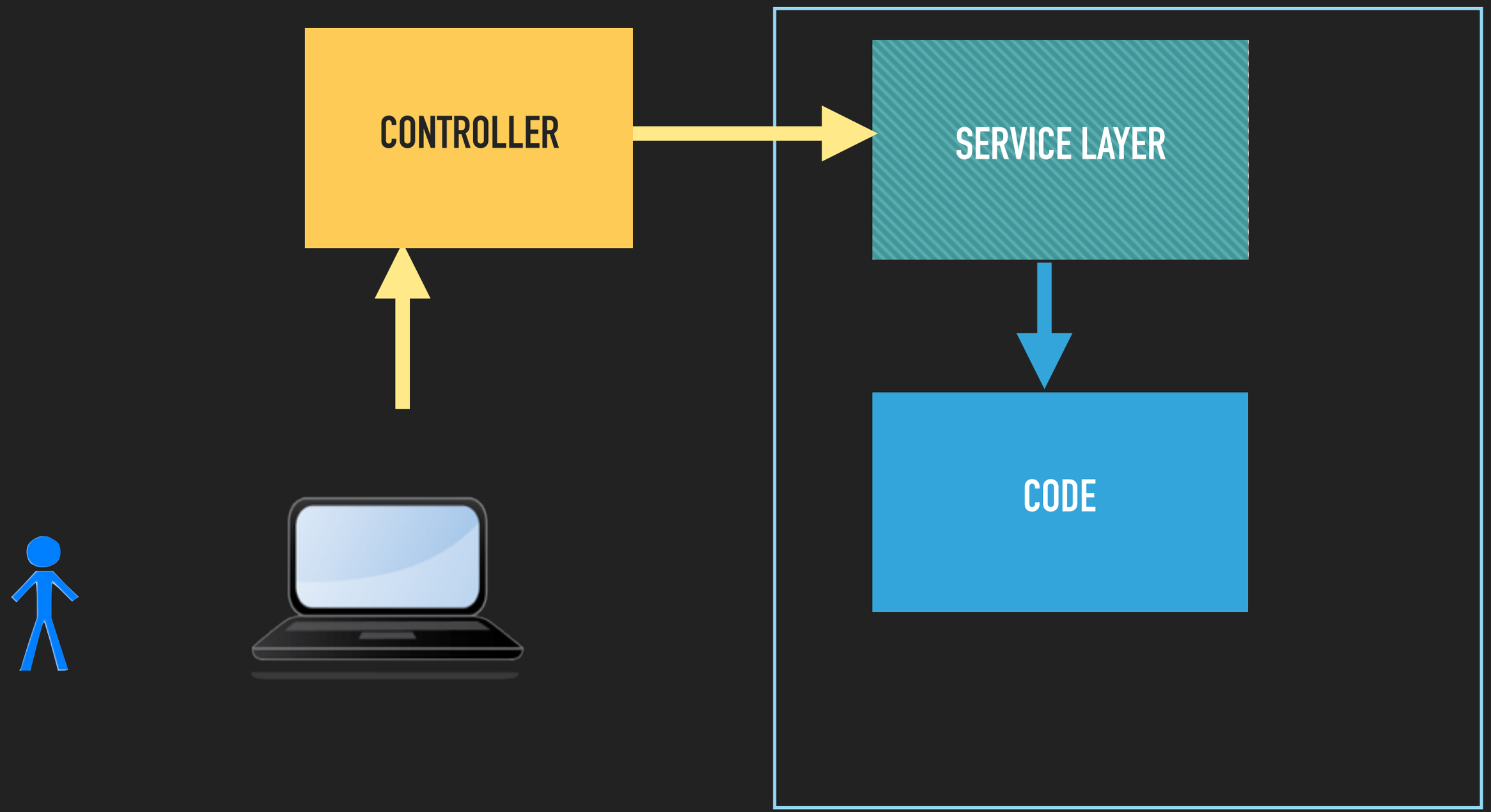
CONTROLLER

CODE

ARCHITECTURE



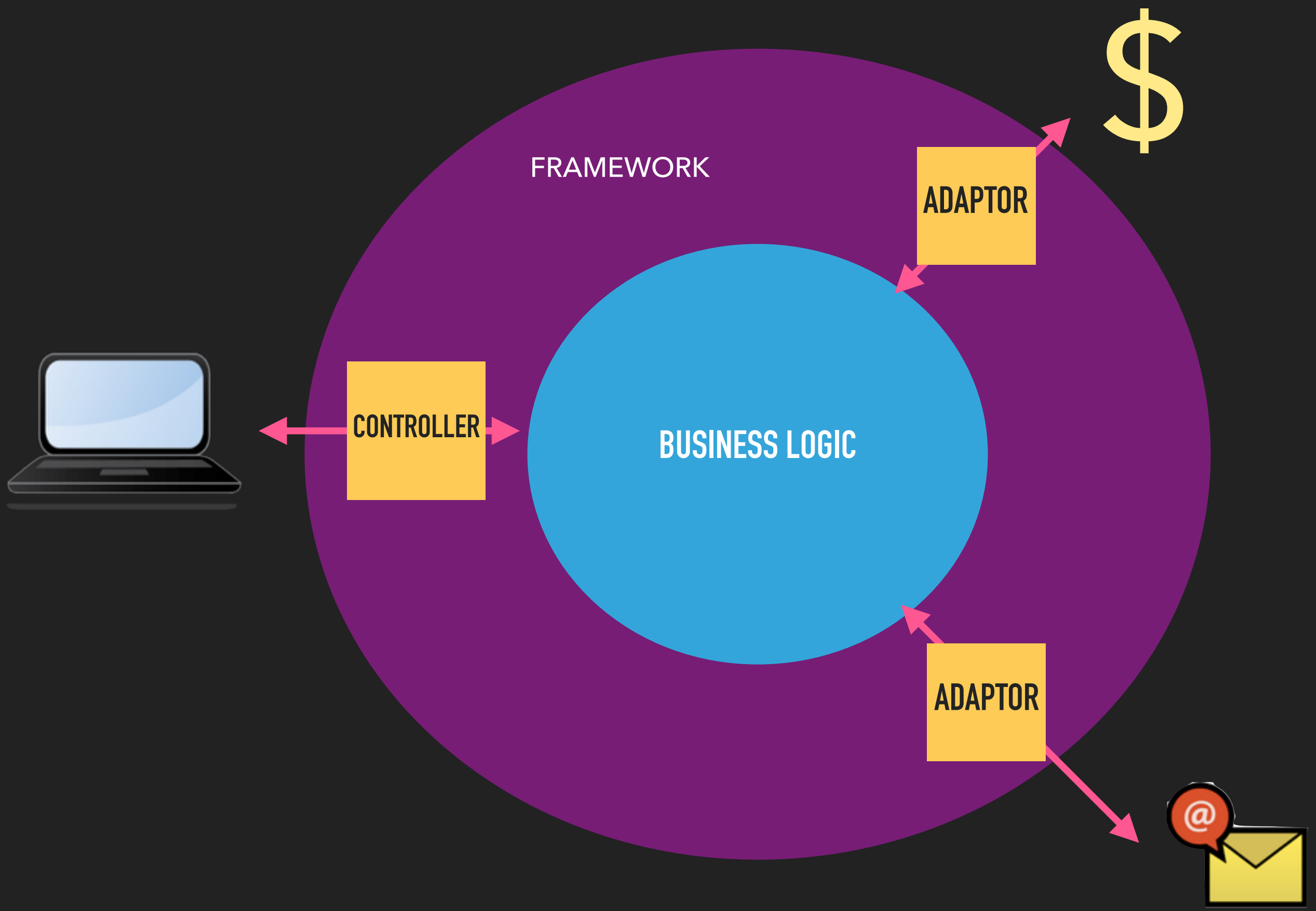
ARCHITECTURE



SERVICE LAYER

```
interface PasswordService
{
  /**
   * Send user link to reset their password
   */
  public function requestPasswordReset($emailAddress);

  /**
   * Reset password from link
   */
  public function resetPassword($token, $newPassword): bool;

  /**
   * Normal password reset
   */
  public function updatePassword($user, $newPassword): bool;

}
```

ARCHITECTURE



ARCHITECTURE

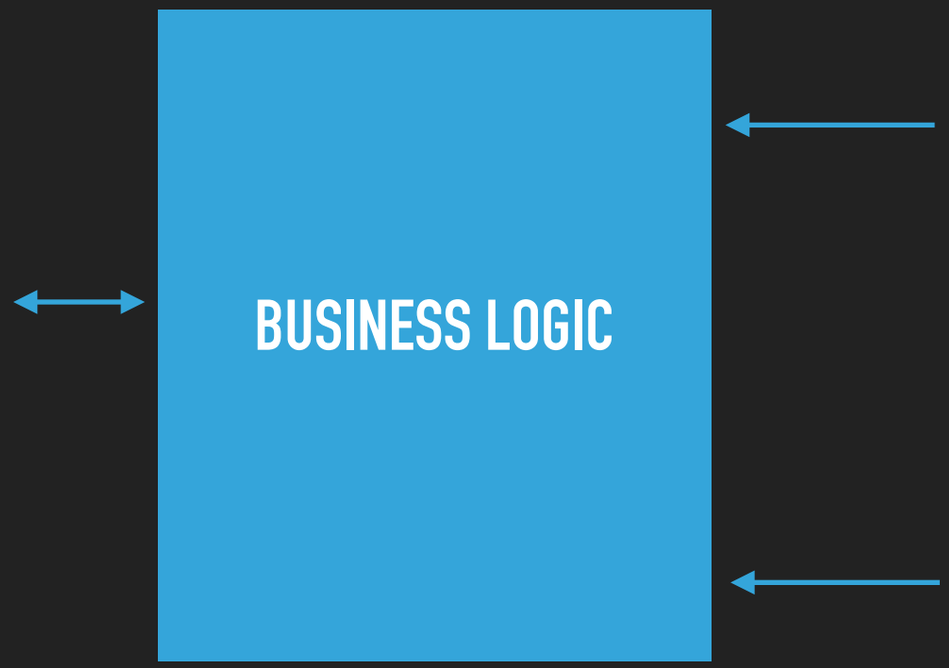
CONTROLLER

BUSINESS LOGIC

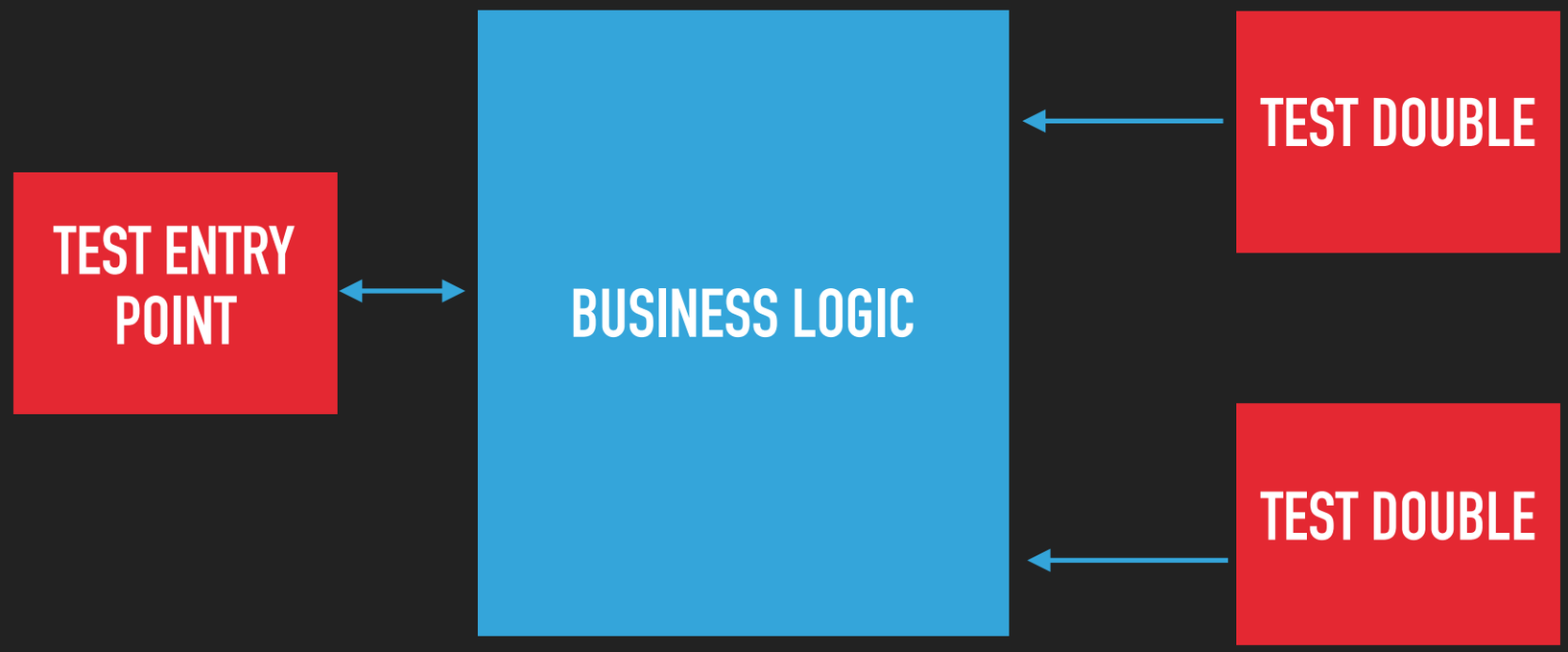
PAYMENT GATEWAY

EMAIL GATEWAY

ARCHITECTURE



ARCHITECTURE



PASSWORD RESET TEST 1

```
class PasswordResetTest extends TestCase
{

  public function testPasswordReset() {

    $user = … create a new user …

    $success = $this->passwordService
        ->requestPasswordReset($user->getEmail());

    $this->assertTrue($success);

    …
```

PASSWORD RESET TEST 1

```
class PasswordResetTest extends TestCase
{

  public function testPasswordReset() {

    $user = … create a new user …

$success = $this->passwordService
     ->requestPasswordReset($user->getEmail());

$this->assertTrue($success);

…
```

PASSWORD RESET TEST 1

```
class PasswordResetTest extends TestCase
{

  public function testPasswordReset() {

    $user = … create a new user …

    $success = $this->passwordService
        ->requestPasswordReset($user->getEmail());

    $this->assertTrue($success);

…
```

PASSWORD RESET TEST 2

…

```
  $emailMessages = $this->fakeEmailGateway->find(
     $user->getEmail(), “PASSWORD_RESET”);

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

$data = $emailMessages[0]->getData();
$token = $data[‘token’];

$success = $passwordService->resetPassword(
     $token, ‘NewPassword1’);

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

PASSWORD RESET TEST 2

...

```
$emailMessages = $this->fakeEmailGateway->find(
    $user->getEmail(), “PASSWORD_RESET”);

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

$data = $emailMessages[0]->getData();
$token = $data[‘token’];

$success = $passwordService->resetPassword(
    $token, ‘NewPassword1’);

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

PASSWORD RESET TEST 2

…

```
$emailMessages = $this->fakeEmailGateway->find(
    $user->getEmail(), “PASSWORD_RESET”);

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

$data = $emailMessages[0]->getData();
$token = $data[‘token’];

$success = $passwordService->resetPassword(
    $token, ‘NewPassword1’);

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

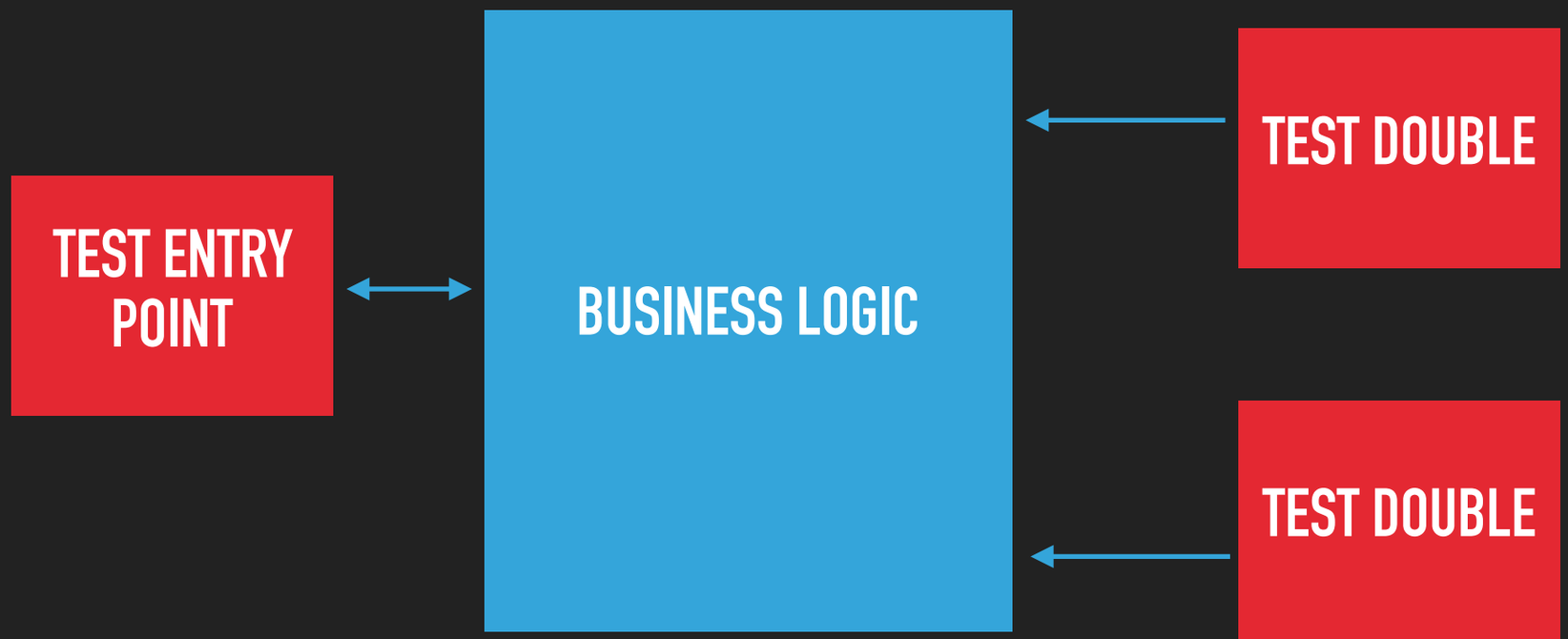
PASSWORD RESET TEST 2

…

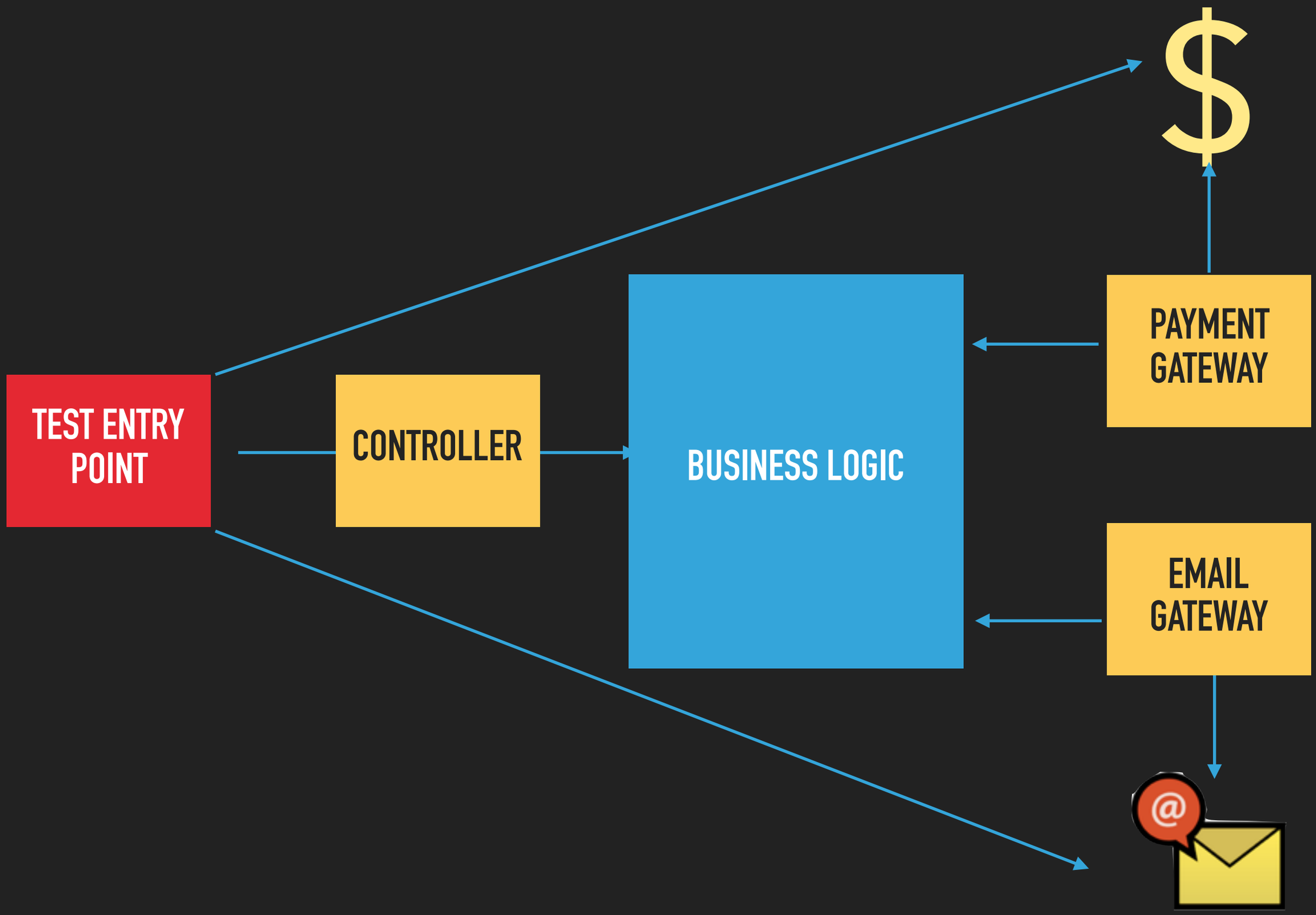
```
$emailMessages = $this->fakeEmailGateway->find(
    $user->getEmail(), “PASSWORD_RESET”);

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

$data = $emailMessages[0]->getData();
$token = $data[‘token’];

$success = $passwordService->resetPassword(
    $token, ‘NewPassword1’);

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

STOP AND ADMIRE

ARCHITECTURE



ARCHITECTURE



ARCHITECTURE IS VERY IMPORTANT

- High correlation between easy to test and good architecture.
- A code base isn't difficult to test, it's poorly architected.

- Holy trinity
  - Lower maintenance
  - Faster
  - Lower coverage

RETURNING TO OUR PASSWORD VALIDATOR: 1

```
public function testUpdatePassword() {

 $user = … create a new user with password Passw1rd …

 // Update password 3 times
$passwordService->updatePassword($user, ‘Passw2rd’);
$passwordService->updatePassword($user, ‘Passw3rd’);
$passwordService->updatePassword($user, ‘Passw4rd’);

 …
```

RETURNING TO OUR PASSWORD VALIDATOR: 2

```
…
    // Update to a recent password
    $success = $passwordService->updatePassword(
            $user, 'Passw1rd');
    $this->assertFalse($success);

    // Update to a non recent password
    $success = $passwordService->updatePassword(
            $user, 'Passw5rd');
    $this->assertTrue($success);

    // Update to password used over 5 times ago
    $success = $passwordService->updatePassword(
            $user, 'Passw1rd');
    $this->assertTrue($success);
```

RETURNING TO OUR PASSWORD VALIDATOR: 2

…

```
// Update to a recent password
$success = $passwordService->updatePassword(
        $user, 'Passw1rd');
$this->assertFalse($success);

// Update to a non recent password
$success = $passwordService->updatePassword(
        $user, 'Passw5rd');
$this->assertTrue($success);

// Update to password used over 5 times ago
$success = $passwordService->updatePassword(
        $user, 'Passw1rd');
$this->assertTrue($success);
```

RETURNING TO OUR PASSWORD VALIDATOR: 2

```
…
    // Update to a recent password
    $success = $passwordService->updatePassword(
            $user, 'Passw1rd');
    $this->assertFalse($success);

    // Update to a non recent password
    $success = $passwordService->updatePassword(
            $user, 'Passw5rd');
    $this->assertTrue($success);

    // Update to password used over 5 times ago
    $success = $passwordService->updatePassword(
            $user, 'Passw1rd');
    $this->assertTrue($success);
```

RETURNING TO OUR PASSWORD VALIDATOR: 2

```
…
    // Update to a recent password
    $success = $passwordService->updatePassword(
            $user, 'Passw1rd');
    $this->assertFalse($success);

    // Update to a non recent password
    $success = $passwordService->updatePassword(
            $user, 'Passw5rd');
    $this->assertTrue($success);

    // Update to password used over 5 times ago
    $success = $passwordService->updatePassword(
            $user, 'Passw1rd');
    $this->assertTrue($success);
```

THE BIG INTEGRATION TEST

CONSTRUCTING OUR USER OBJECT

```
$user … 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

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

HAND BUILDING

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

$userService->completeRegistration(
        $user->getConfirmationToken);
```

HAND BUILDING

```
$user = $this->userService->registerUser(
                    "dave@lampbristol.com",
                    "Dave",
                    "Passw0rd");

$userService->completeRegistration(
        $user->getConfirmationToken);
```

SEEDING A DATABASE

```
users:
  - name: Dave
    email: dave@lampbristol.com
    password: Passw1rd

  - name: Sarah
    email: sarah@example.com
    password: Passw5rd
```

SEEDING A DATABASE

```
users:
  - name: Dave
    email: dave@lampbristol.com
    password: Password

  - name: Sarah
    email: sarah@example.com
    password: Passw5rd
```

OBJECT MOTHER

```
$user = $this->userObjectMother->getDave();

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

OBJECT MOTHER: IMPLEMENTATION

```
class UserObjectMother {

    …

    public function getDave(): User {
        $user = $userService->registerUser(
                    “dave@lampbristol.com”,
                    “Dave”,
                    “Passw0rd”);


        return $user;
    }
```

OBJECT MOTHER: IMPLEMENTATION

```
class UserObjectMother {

    …

    public function getDave(): User {
        $user = $userService->registerUser(
                    “dave@lampbristol.com”,
                    “Dave”,
                    “PasswOrd”);

        $userService->confirmRegistration(
            $user, $user->getToken());
        return $user;
}
```

TEST BUILDER: 1

```
$userBuilder = $this->getUserBuilder();
$user = $userBuilder->build();

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

USING A TEST BUILDER (2)

```
$userBuilder = $this->getUserBuilder();
$user = $userBuilder
            ->name(“David”)
           ->password(“Passw4rd”)
          ->previousPasswords([
              “Passw1rd”,
              “Passw2rd”,
              “Passw3rd”,
           ])
          ->build();
```

DECOUPLING OUR TESTS FROM CODE UNDER TEST

OBJECT MOTHER AND TEST BUILDER BENEFITS

- Single place where test business object built
  - Easy to find
  - Easy to update
- Defer to other Object Mothers / Test Builders
- Decoupling our tests from the software under test
  - More robust
  - Easier to refactor

TAKE AWAY

USE OBJECT MOTHER AND TEST BUILDER PATTERNS

- Reduce coupling between test and production code
- Help make your tests more resilient to change
- Holy trinity:
  - Lower maintenance
  - Higher coverage

FABULOUS 5 VS AWKWARD DUO VS THE BIG INTEGRATION TEST?

FABULOUS 5 VS

~~AWKWARD DUO~~ VS

THE BIG INTEGRATION TEST?

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

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

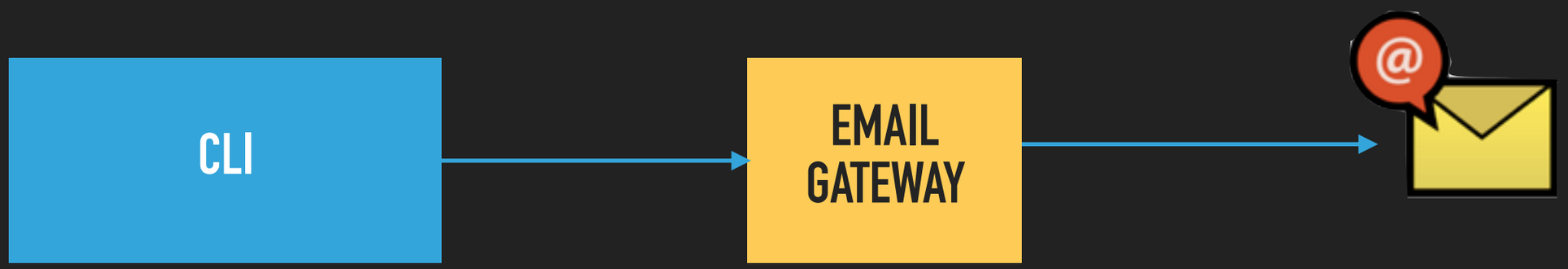
QUESTIONS

https://joind.in/talk/f3d90

BONUS 1

CAN WE AUTOMATE ANYTHING ELSE?

CAN WE AUTOMATE ANYTHING ELSE?



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]
```