The Test Suite Holy Trinity

Dave Liddament

@DaveLiddament

First a sad story….

…. about a dark time

I still have nightmares

Why this talk?


Dave Liddament
@daveliddament
Organise PHP-SW and Bristol PHP Training

Back to the nightmare…

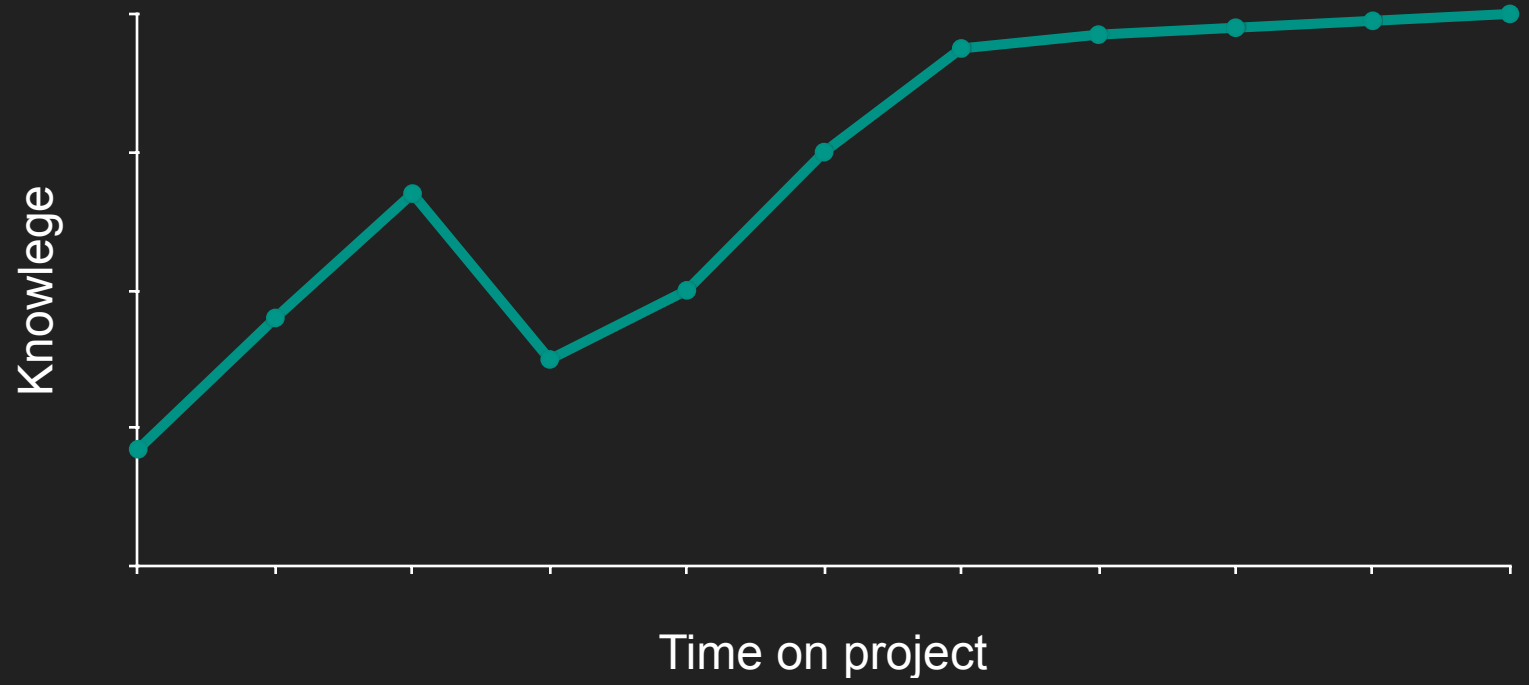
#1 I didn’t know much about developing high quality software

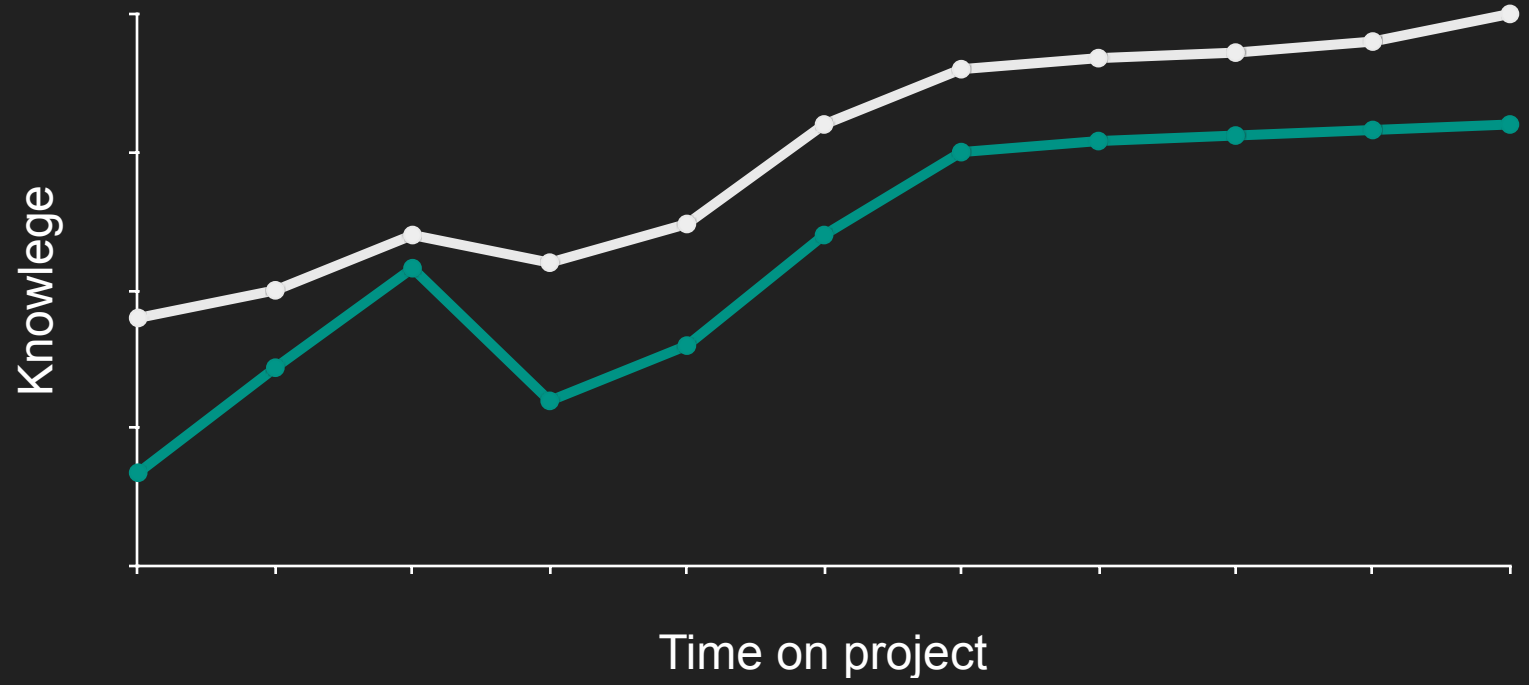
#2 Copy someone who does know about developing high quality software

We need tests

We need a test suite

Ability to refactor is important


Knowlege
Time on project


Knowlege
Time on project

A quick recap…

A test suite…

#1 Proves code works

#2 Stops regression

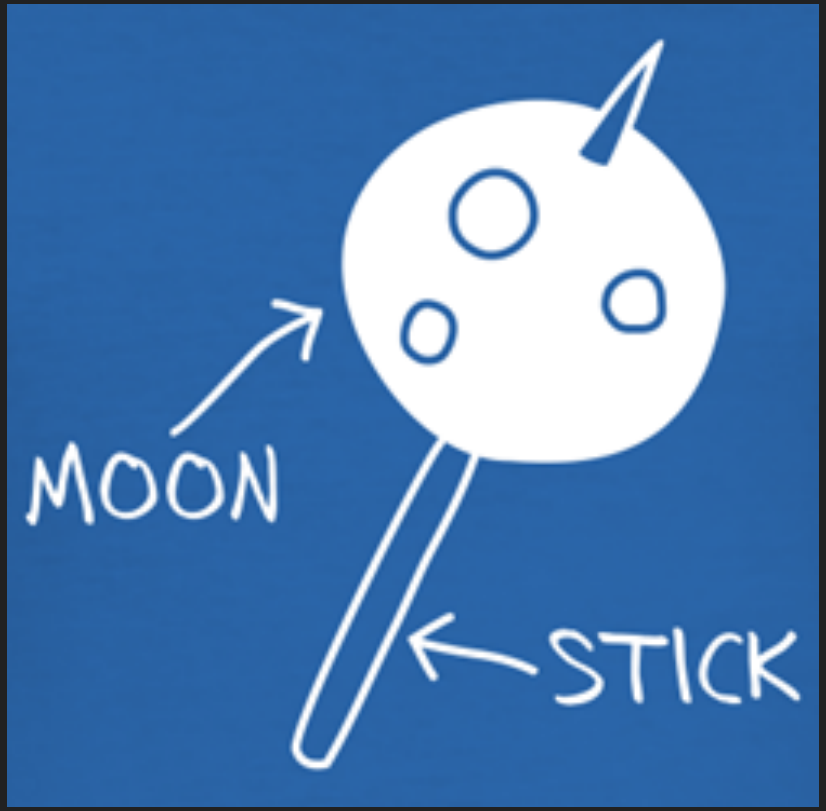
#3 Enables refactoring

The ideal test suite…

Fast to execute

High coverage

Low maintenance


MOON
STICK

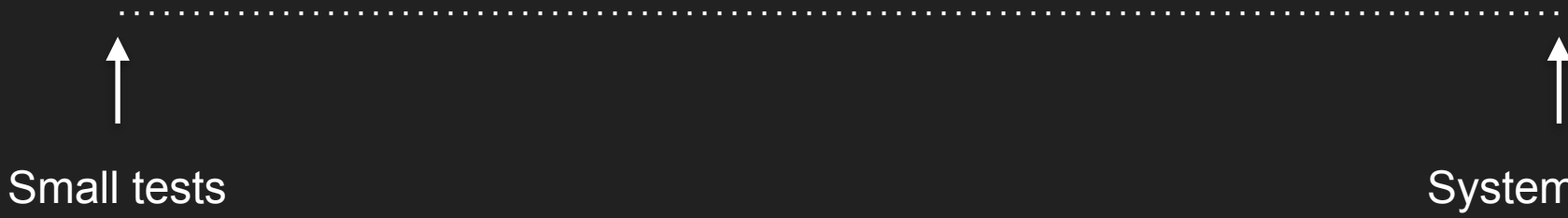
The Holy Trinity…

#1 Fast to execute

#2 High coverage

#3 Low maintenance

Testing Continuum



Small test example

```
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
```

Test cases required

- Valid password:
  - Passw0rd
- Invalid passwords
  - Too short: Passw0r
  - No digit: Password
  - No upper case: psssw0rd
  - No lower case: PASSW0RD

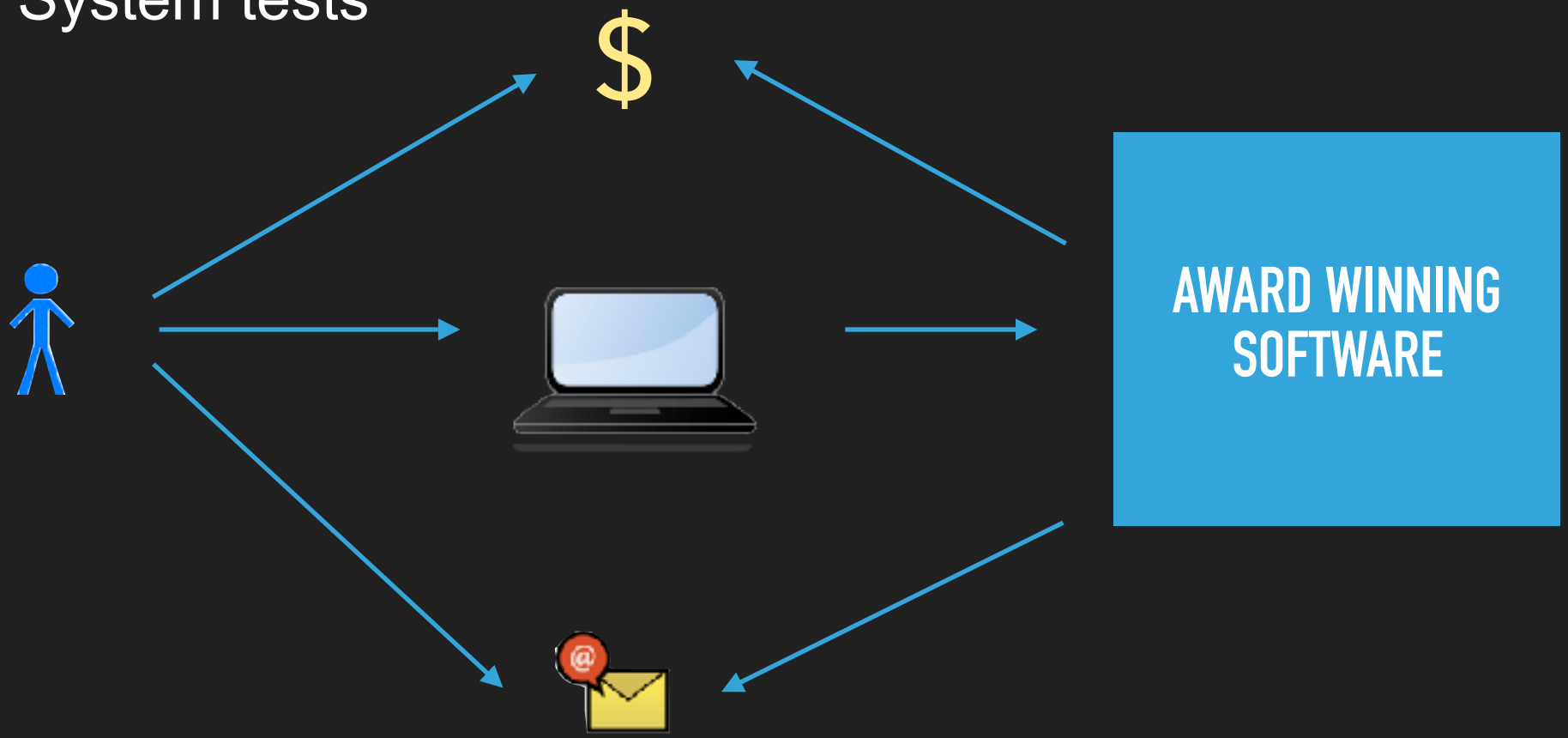
```
class PasswordValidatorTest extends TestCase
{

    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);
    }
```

Take away:
Unit test this kind of logic

System tests



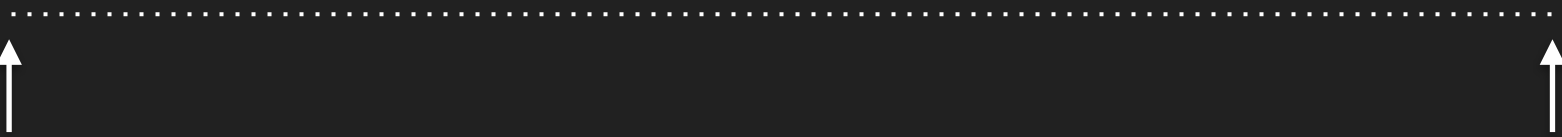
Testing continuum

Testing continuum
#1 Fast to execute

Testing Continuum: Automation

All

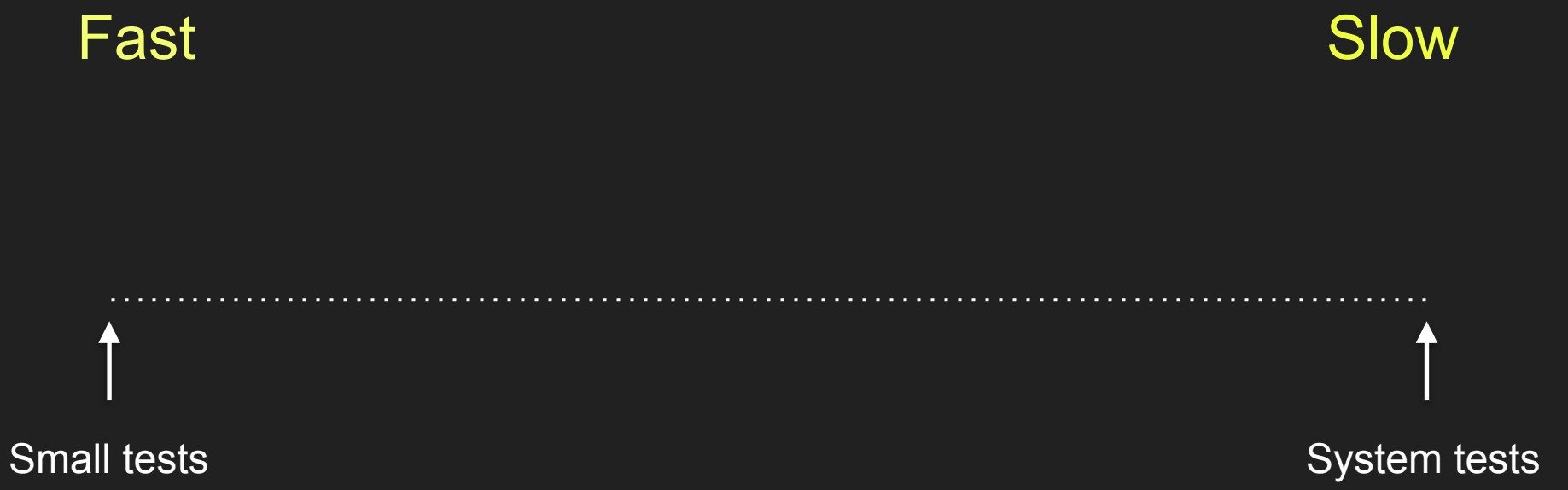
Some



Small tests

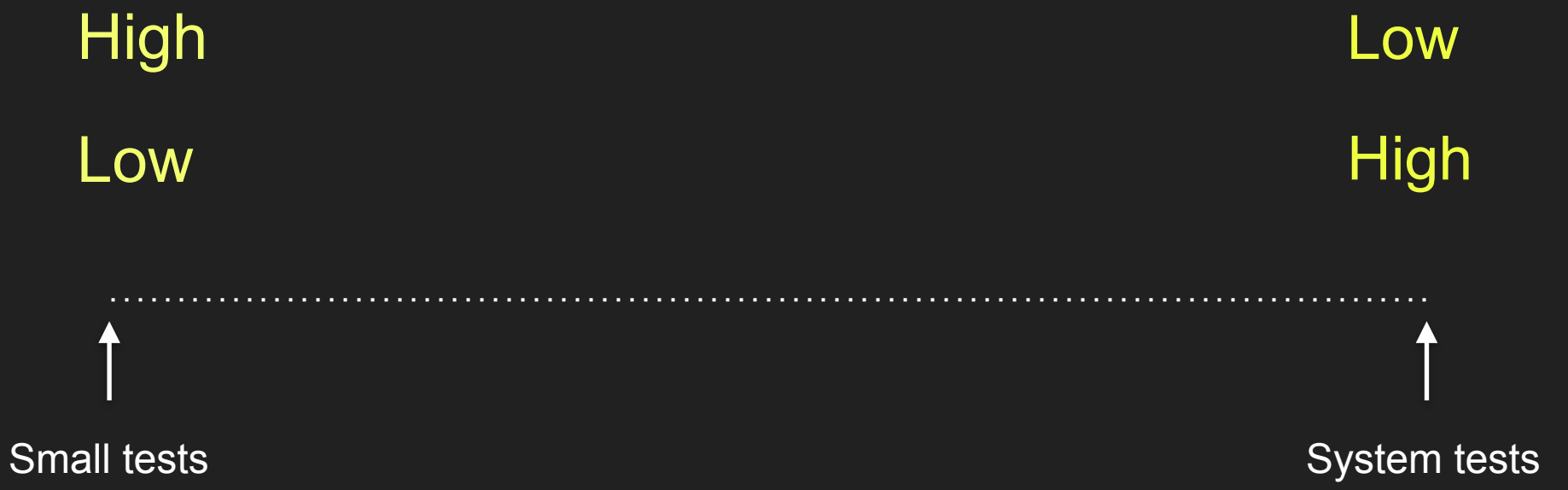
System tests

Testing Continuum: Speed of execution



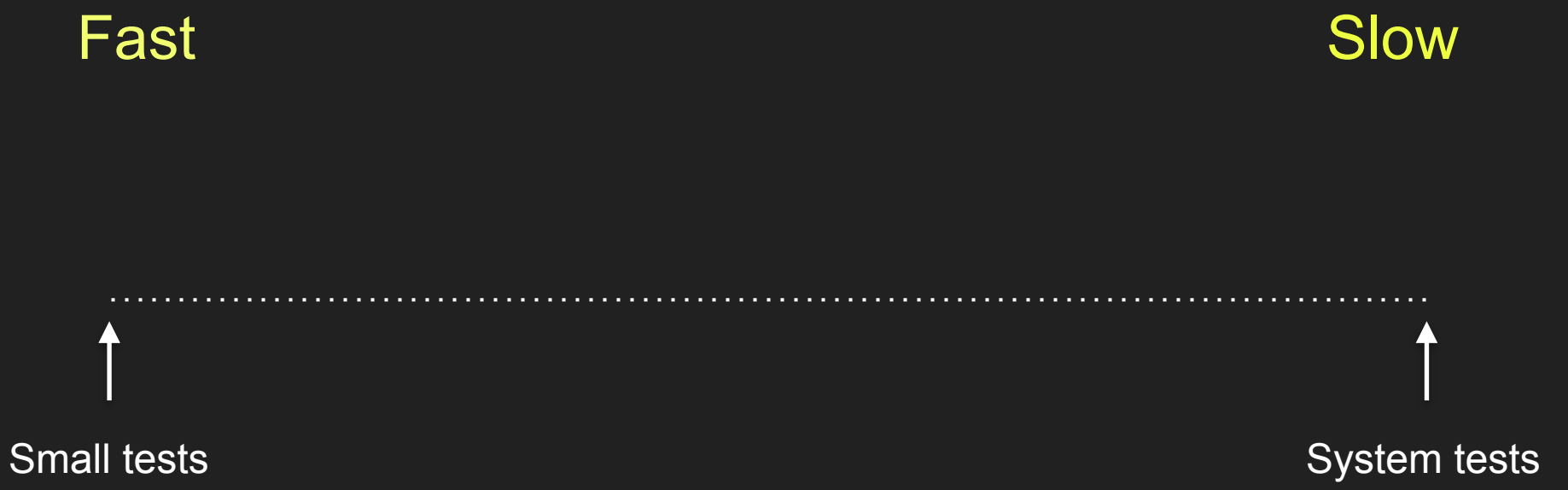
Testing continuum
#2 High coverage

Testing Continuum: Coverage

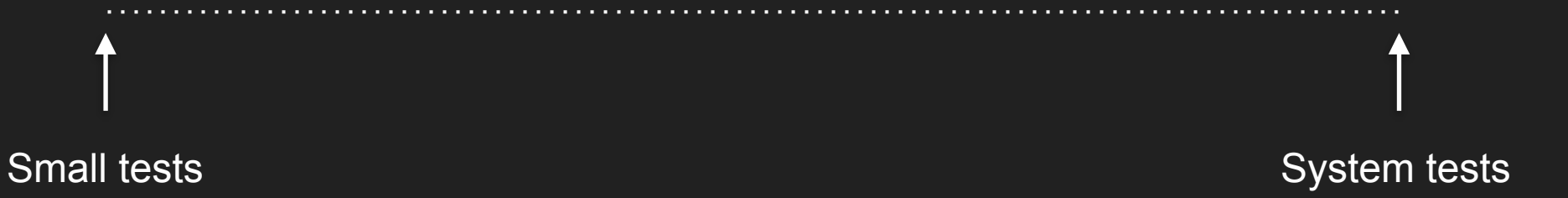


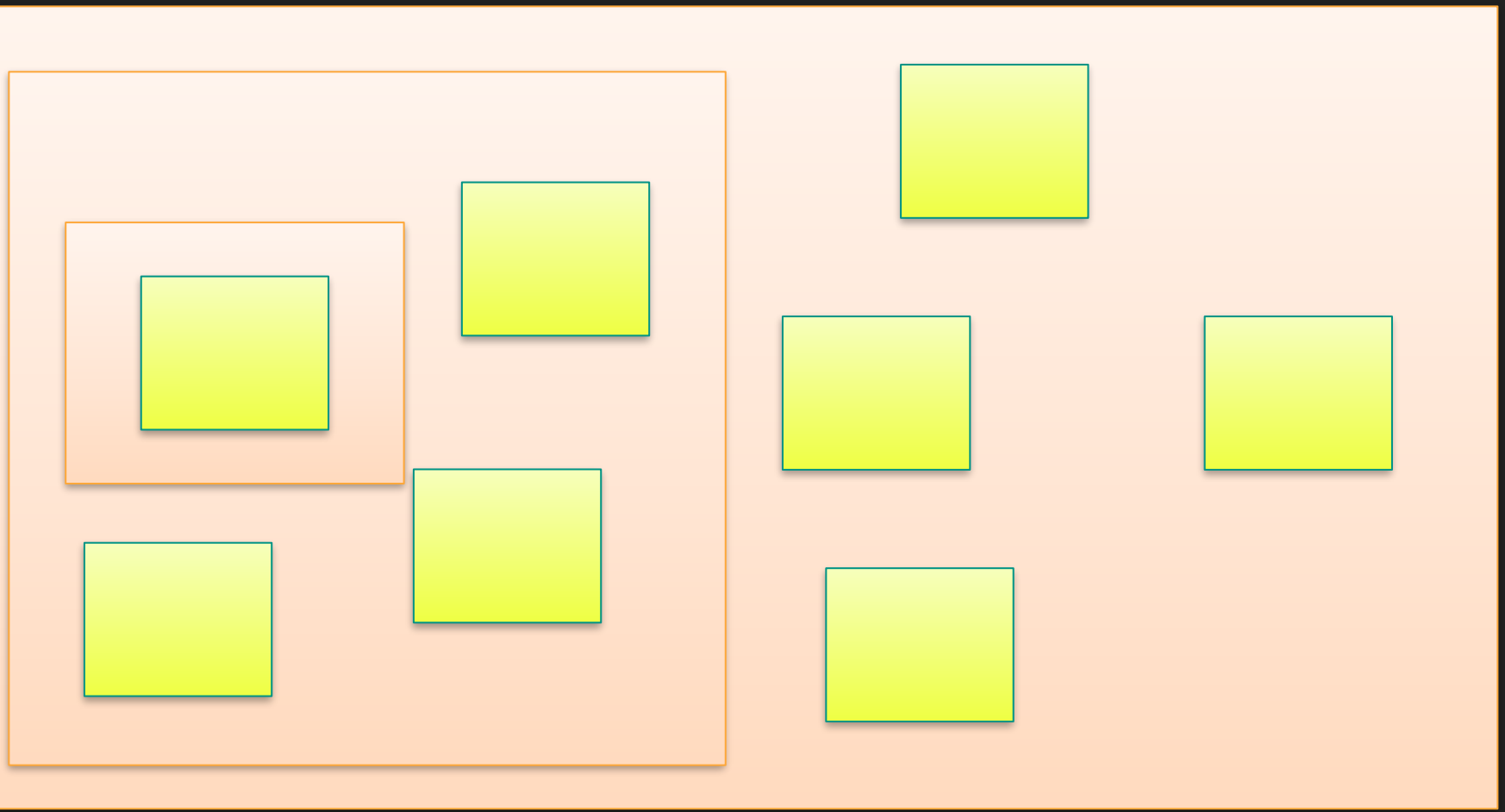
Testing continuum
#3 Low maintenance

Testing Continuum: Speed of writing



Testing Continuum: Debug speed

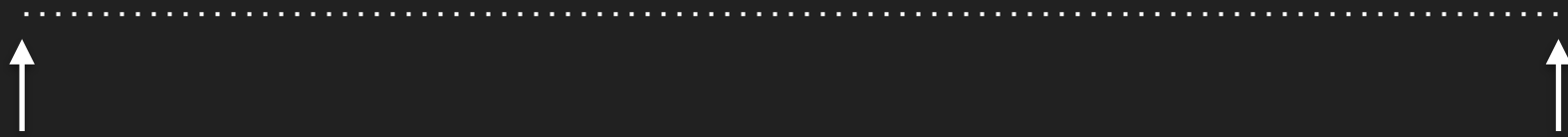




Testing Continuum: Debug speed

Fast

Slow



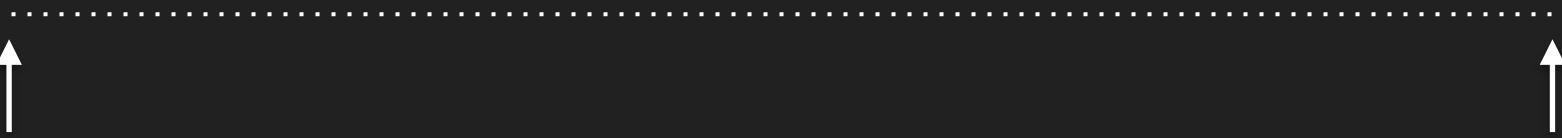
Small tests

System tests

Testing Continuum: Robustness

Robust*

Fragile



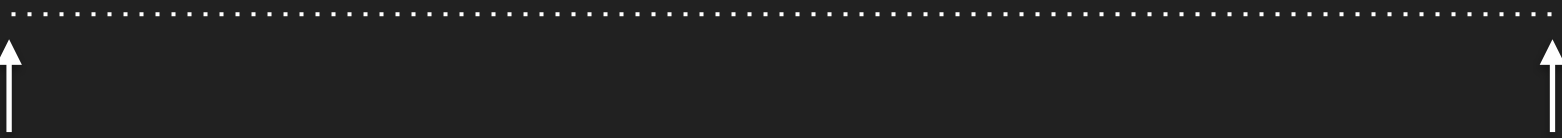
Small tests

System tests

Testing Continuum: Refactoring scope

Small

Large*



Small tests

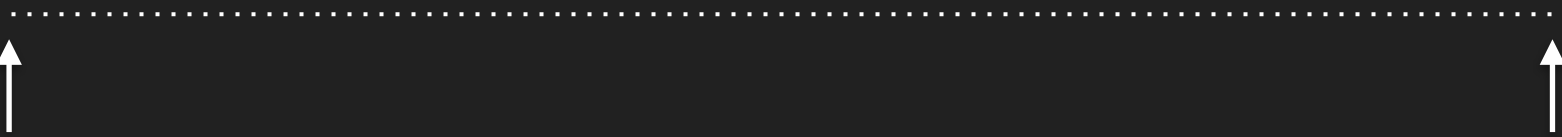
System tests

Other considerations

Testing Continuum: Phew factor

Small

Large



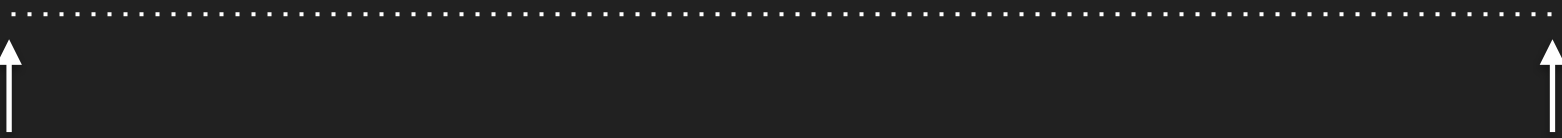
Small tests

System tests

Testing Continuum: Bearing on reality

Not much

Close



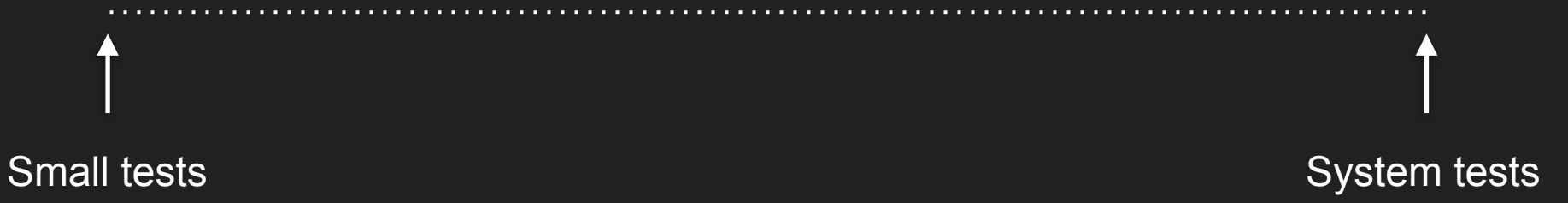
Small tests

System tests

So far nothing too controversial

Where along the testing continuum should we test?

Testing Continuum: Where should we test?

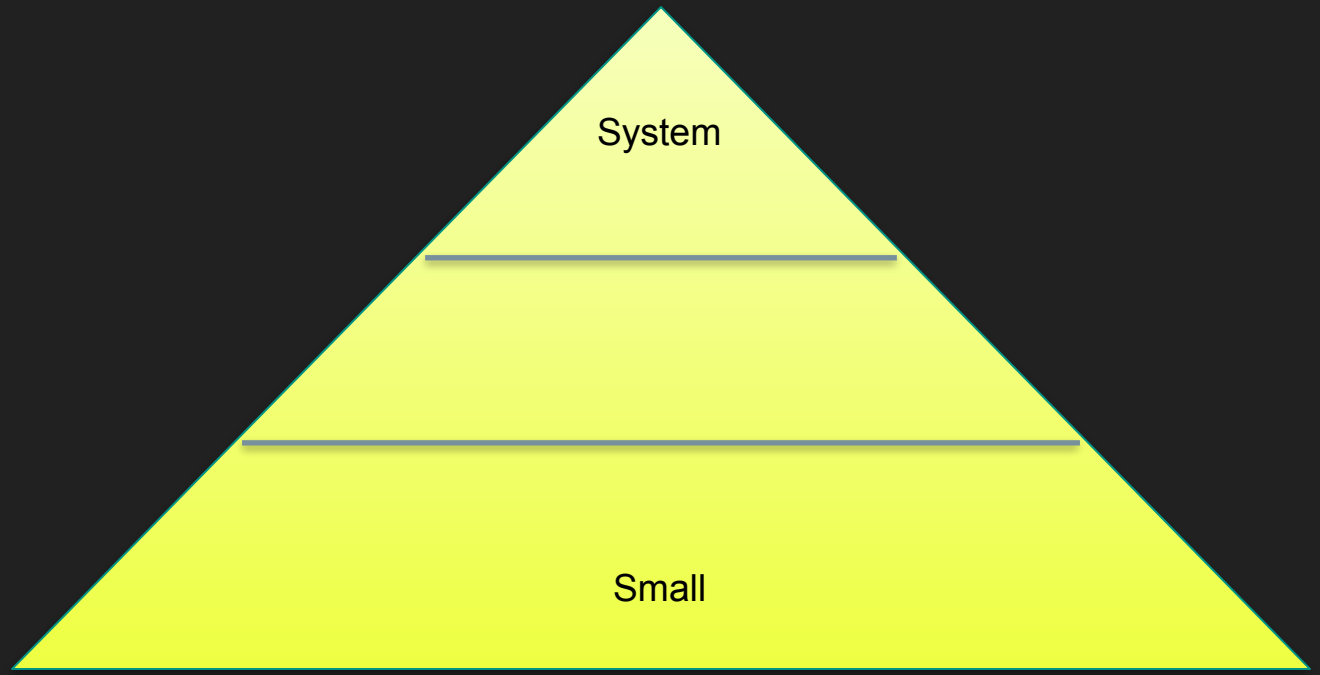


Nothing is black and white

Everything is a compromise

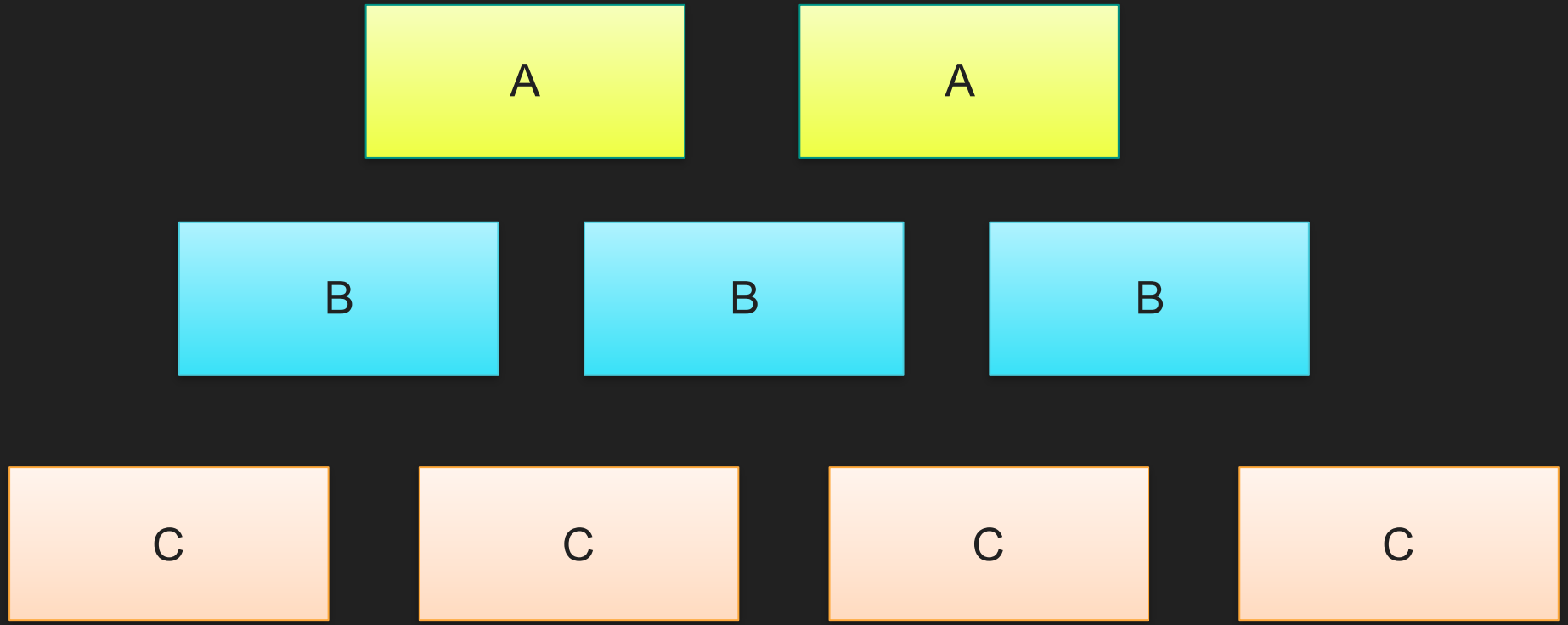
Be pragmatic

Test Pyramid



Test pyramid is still a compromise

Test in layers

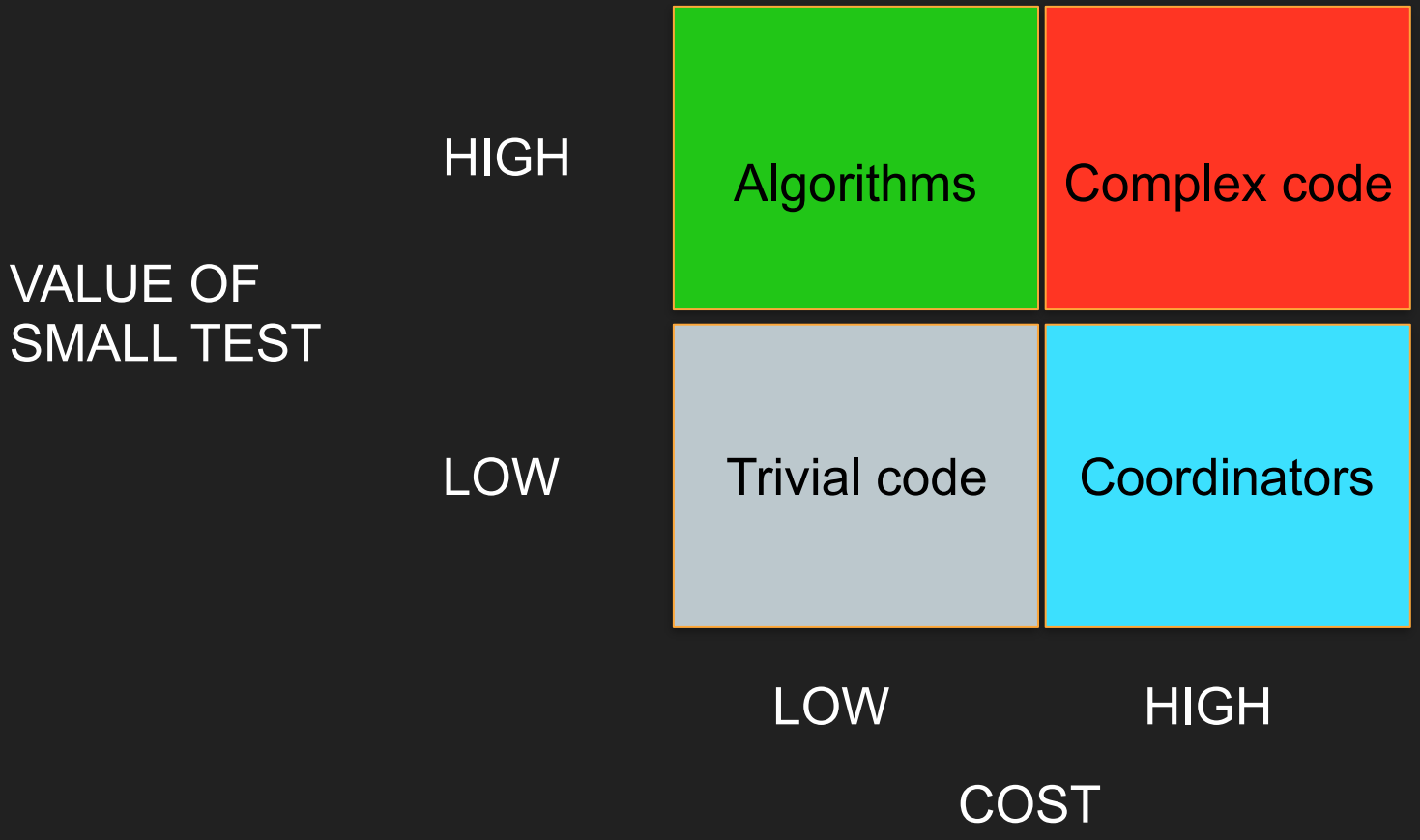


Test in layers - we all do this

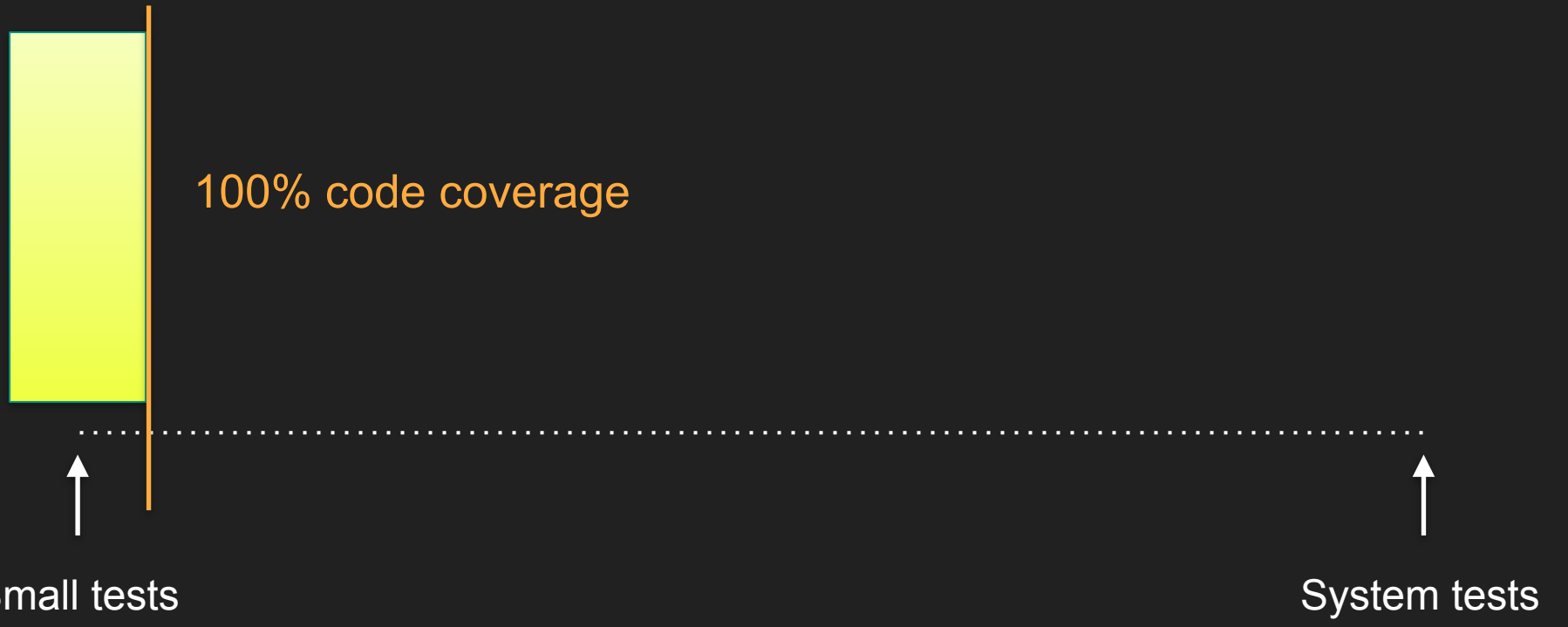
PHP application code

PHP instructions / 3rd party libraries

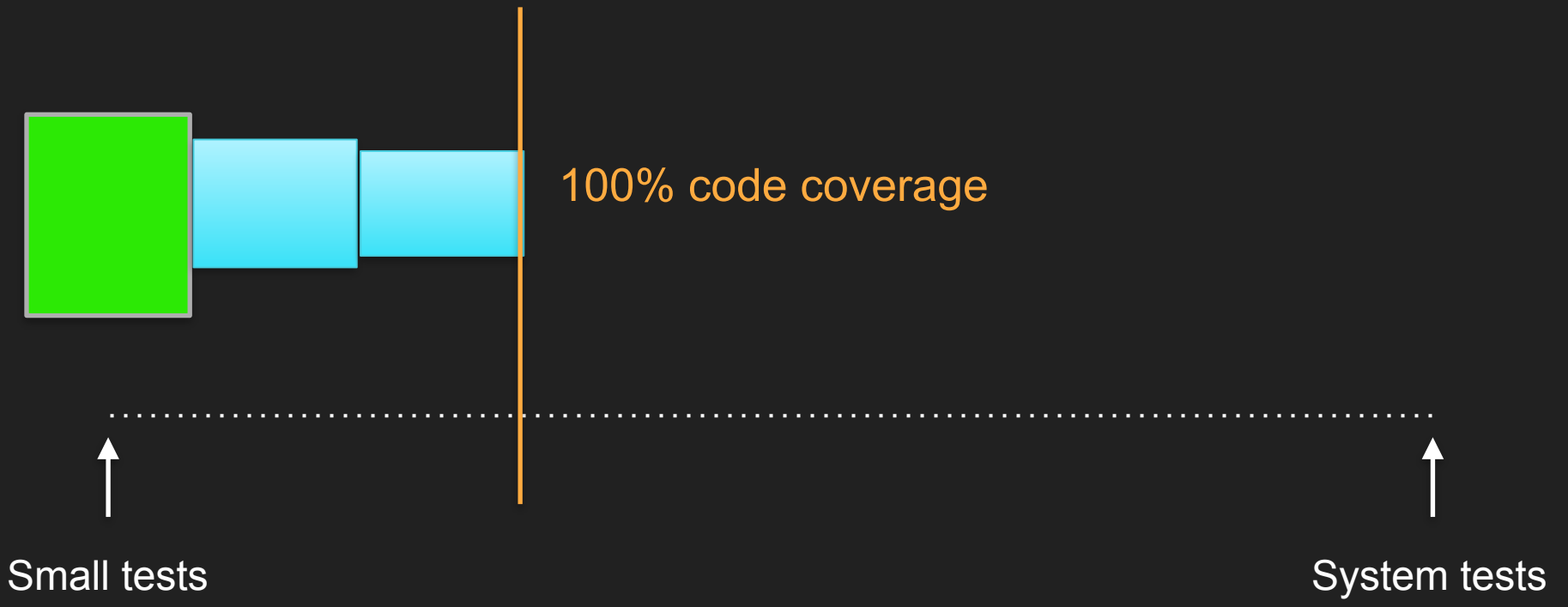
Machine code running on computer


VALUE OF
SMALL TEST
HIGH
LOW
Algorithms
Complex code
Trivial code
Coordinators
LOW
HIGH
COST

Should all production code be ‘unit tested’ ?



Get coverage from more than small test

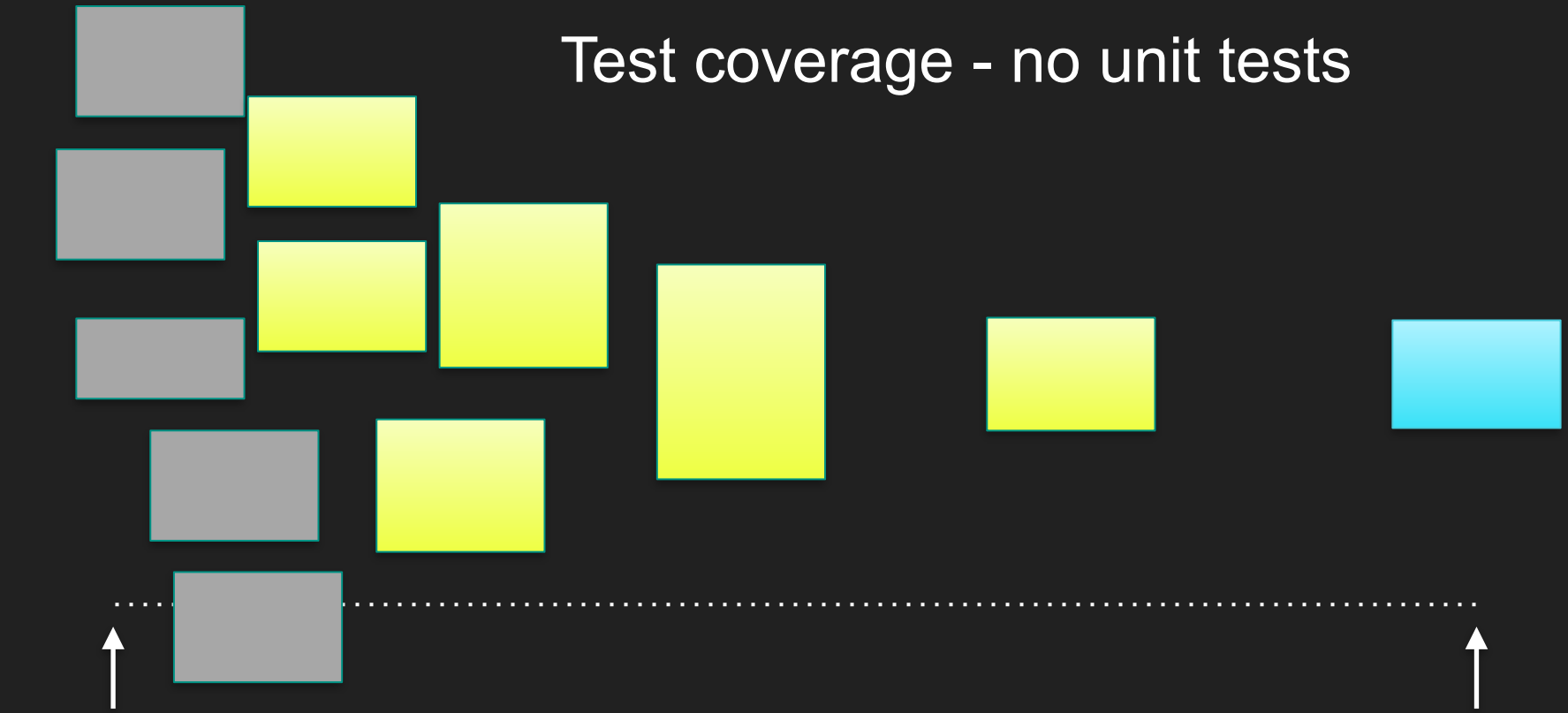


I’m going to transfer £100 to you*

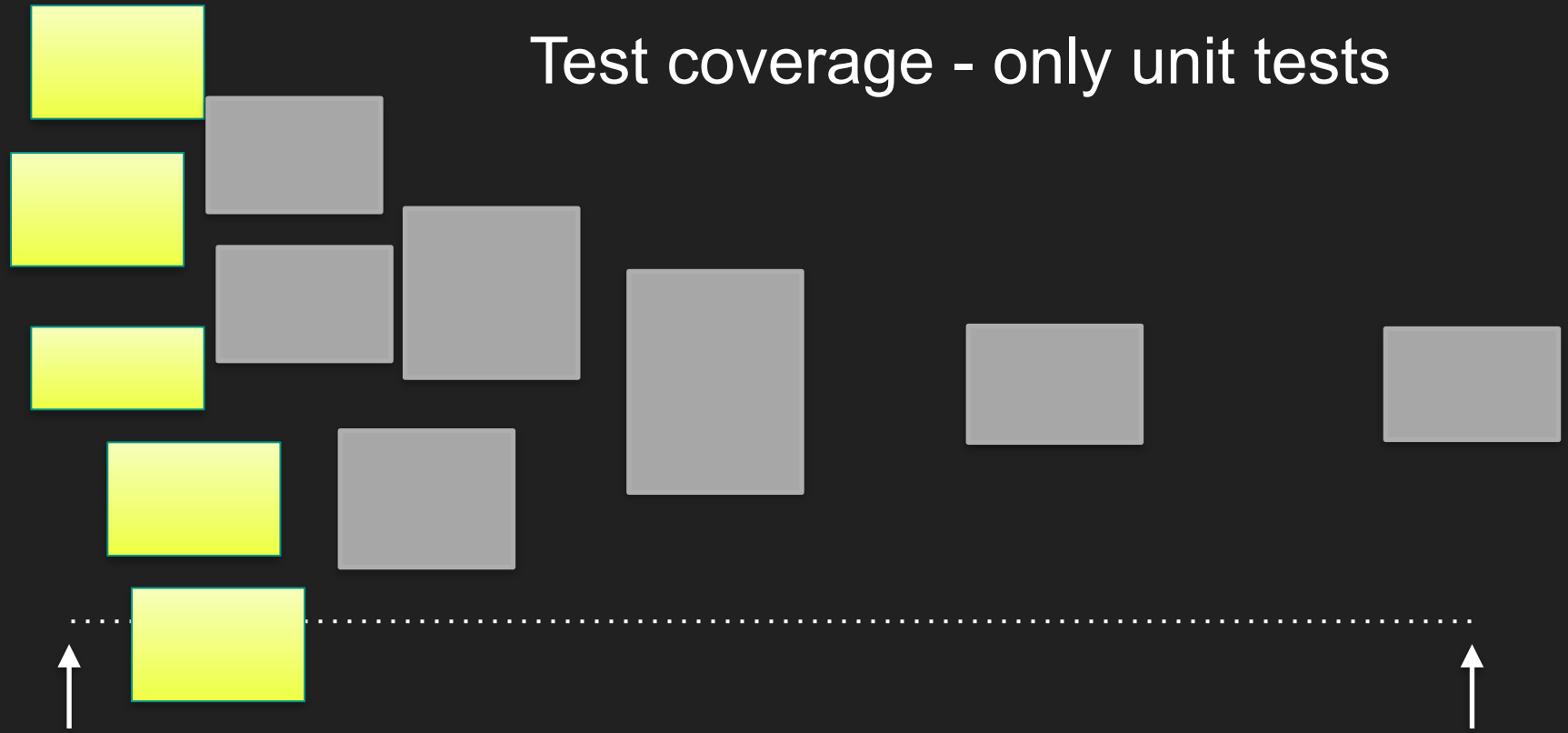
*Assuming everything works

Test coverage

Test coverage - no unit tests



Test coverage - only unit tests



Put the tests where there is highest value

A quick recap…

A test suite…

#1 Proves code works

#2 Stops regression

#3 Enables refactoring

The Holy Trinity…

#1 Fast to execute

#2 High coverage

#3 Low maintenance

Architecture

The codebase isn't
difficult to test,
it's poorly architected

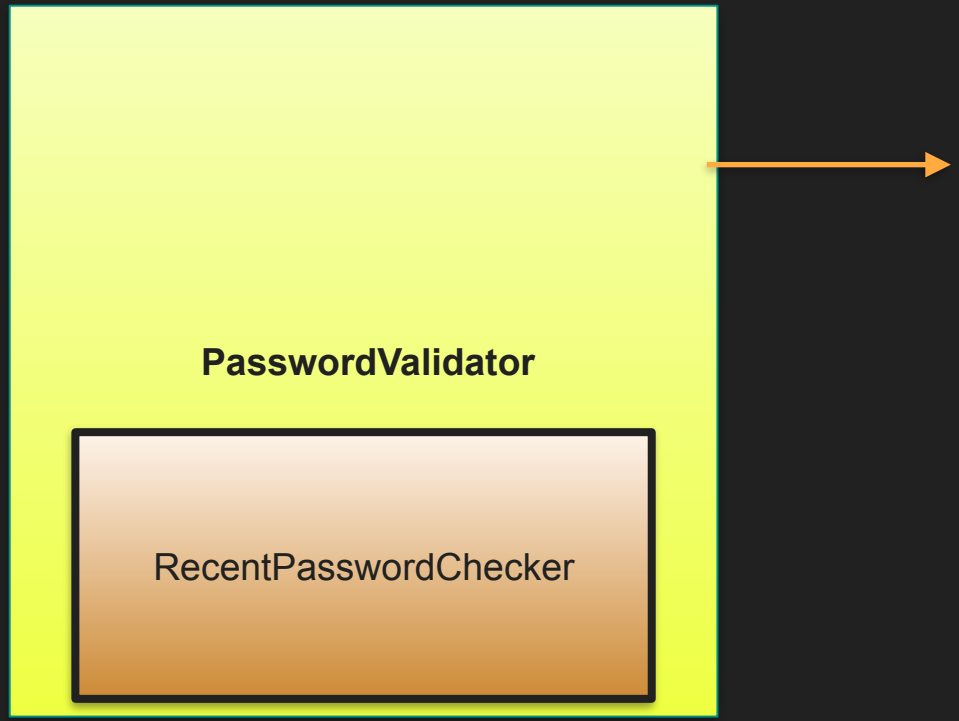
Password Validator

```
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
```

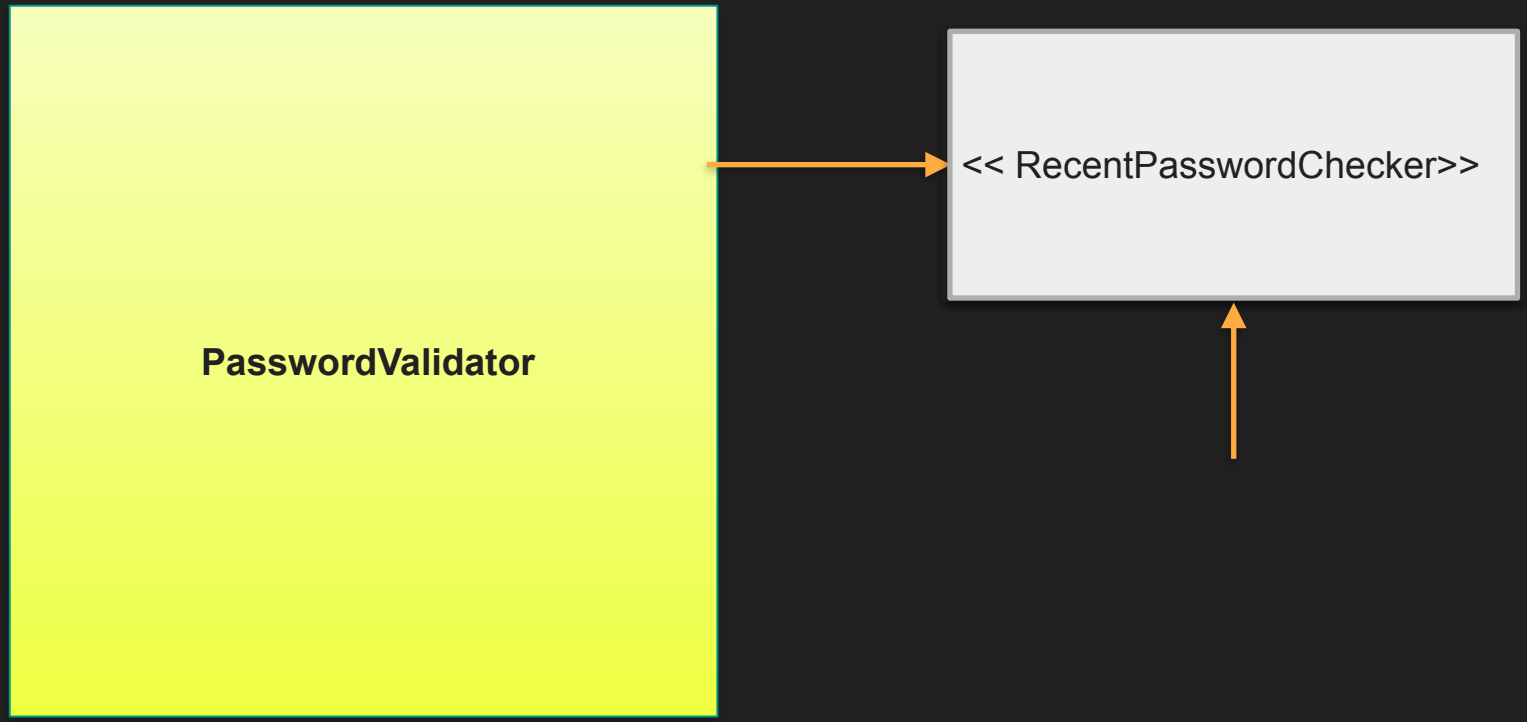
Extended Password Validator

```
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 the previous user's 5 passwords
   */
   public function isValid(string $password, User $user) : bool
```

Architecture: Small tests



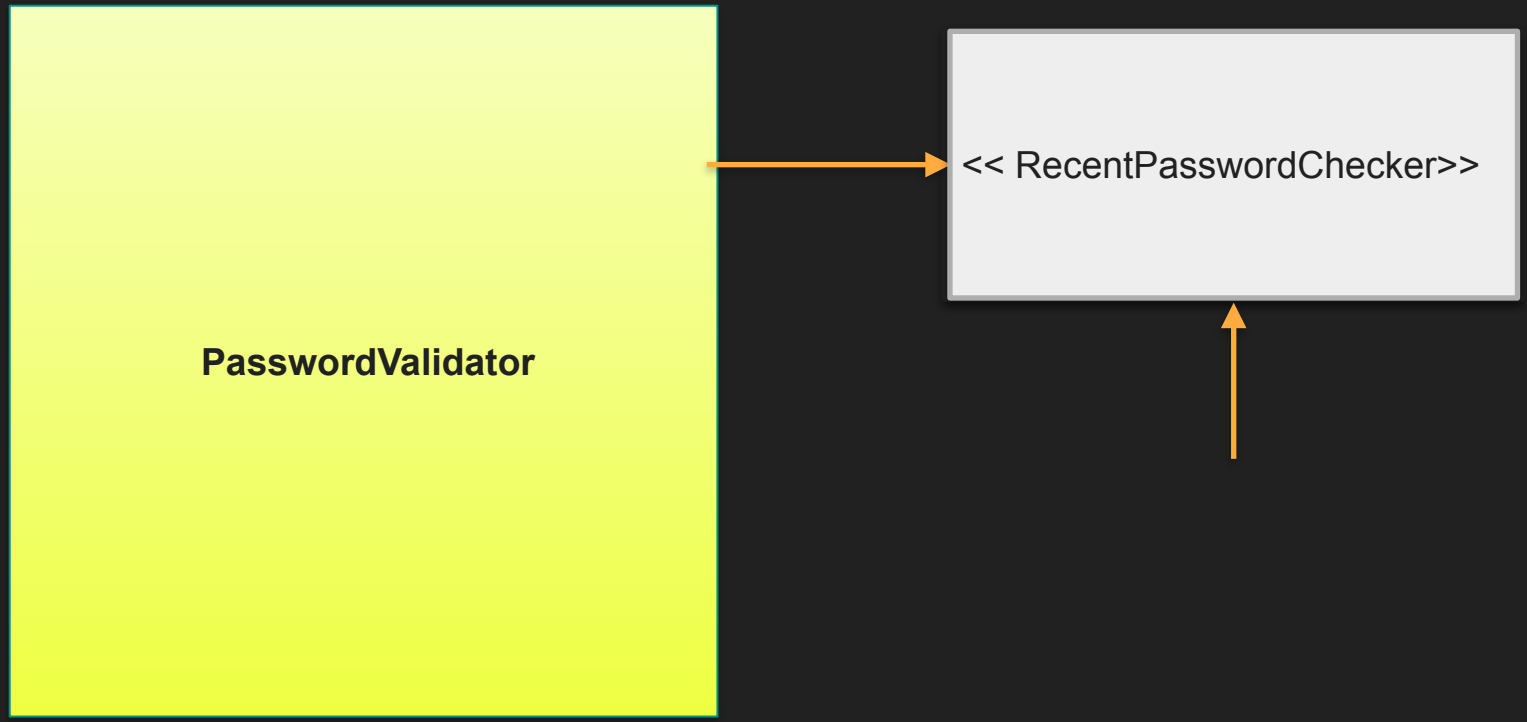
Architecture: Small tests



Password Validator - Checking Previous Passwords

```
interface RecentPasswordChecker
{

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

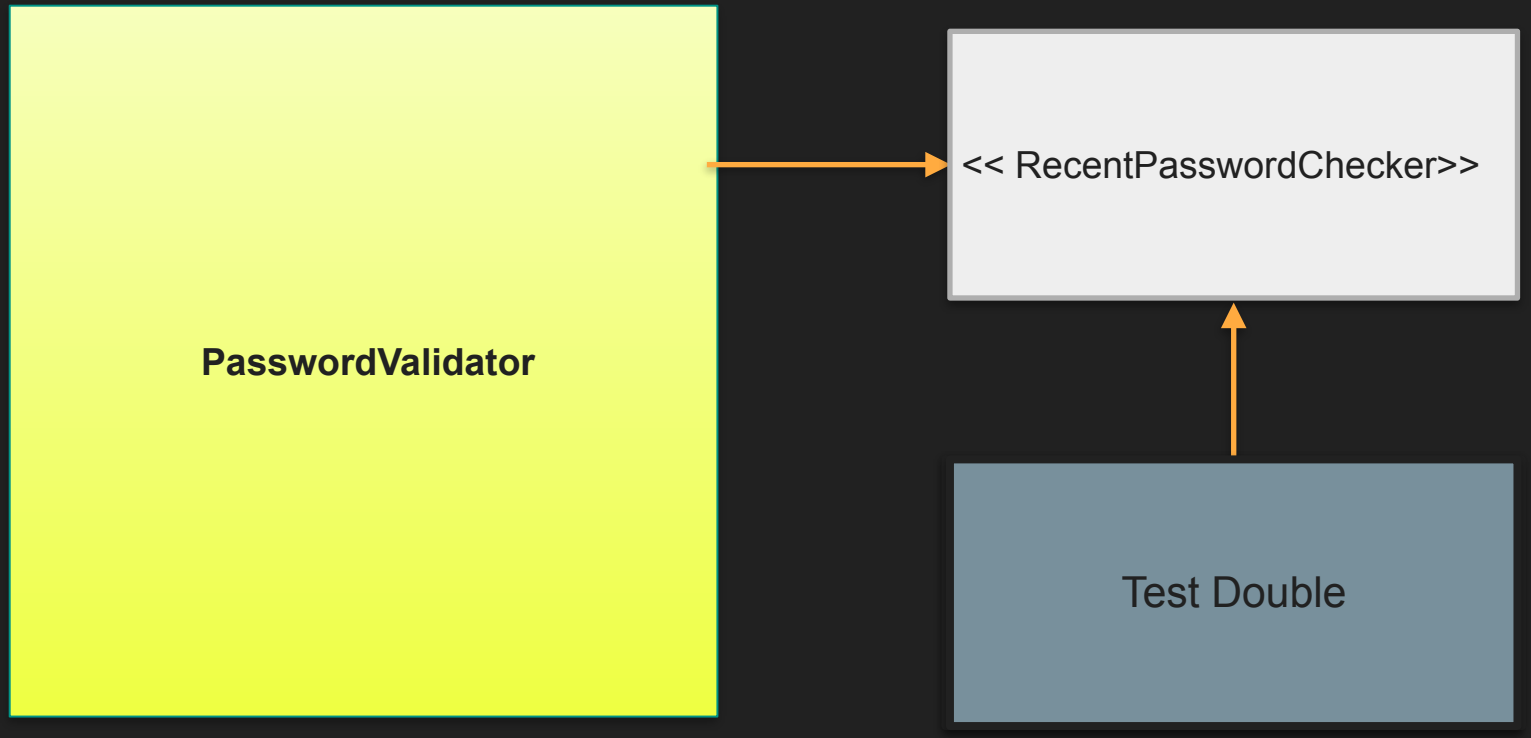
Architecture: Small tests



What do we do with collaborating objects?

- Real version
- Test dummy
  - Stub
  - Mock
  - Fake

Architecture: Small tests



Test double is an approximation

Interface test double must implement

```
interface RecentPasswordChecker
{

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

New tests (1): Not recent password

- Assume we call isValidPassword with Passw0rd
- Assert isValidPassword returns true
- Mock for RecentPasswordChecker
- isRecentPassword called once
- isRecentPassword returns false
- isRecentPassword called with Passw0rd

New tests (2): Recent password

- Assume we call isValidPassword with Passw0rd
- Assert isValidPassword returns false
- Mock for RecentPasswordChecker
- isRecentPassword called once
- isRecentPassword returns true
- isRecentPassword called with Passw0rd

Existing tests?

Password Validator implementation

```
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

- Test inputs as before
- Mock for RecentPasswordChecker
- isRecentPassword called once
- isRecentPassword returns false
- isRecentPassword called with test value

Password Validator implementation refactored

```
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;
  }
```

Our tests start failing

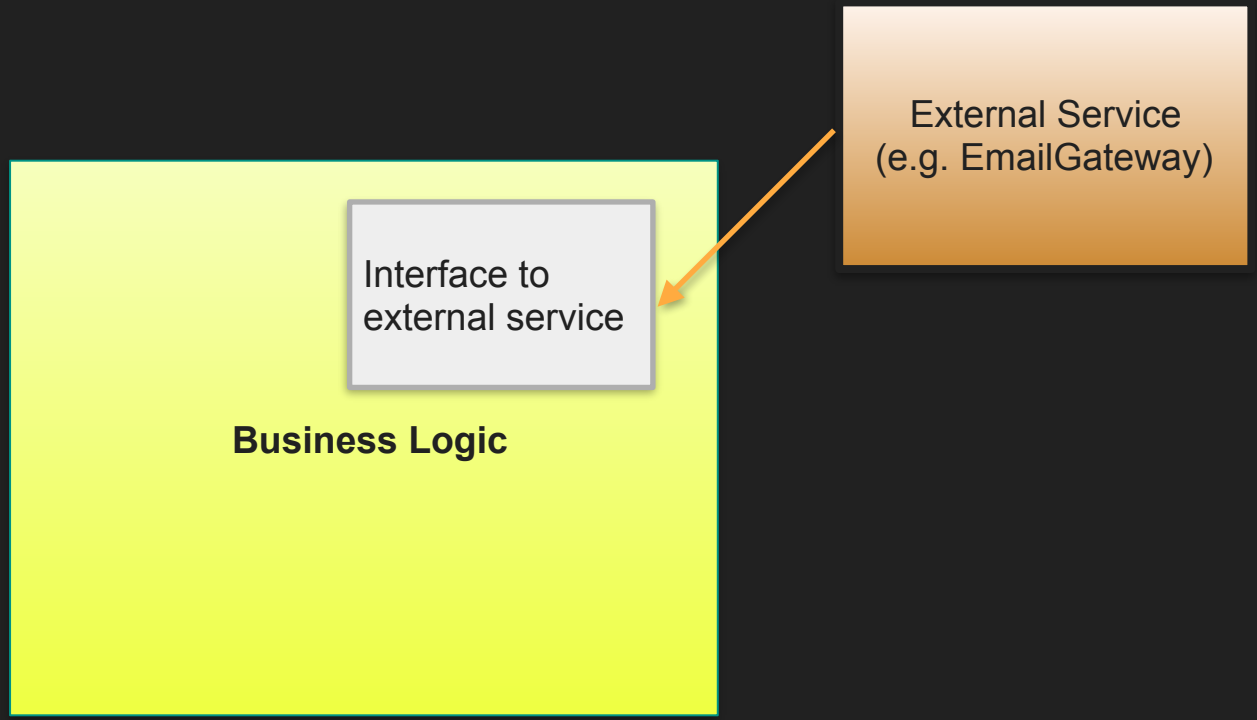
High maintenance test suite (which is bad)

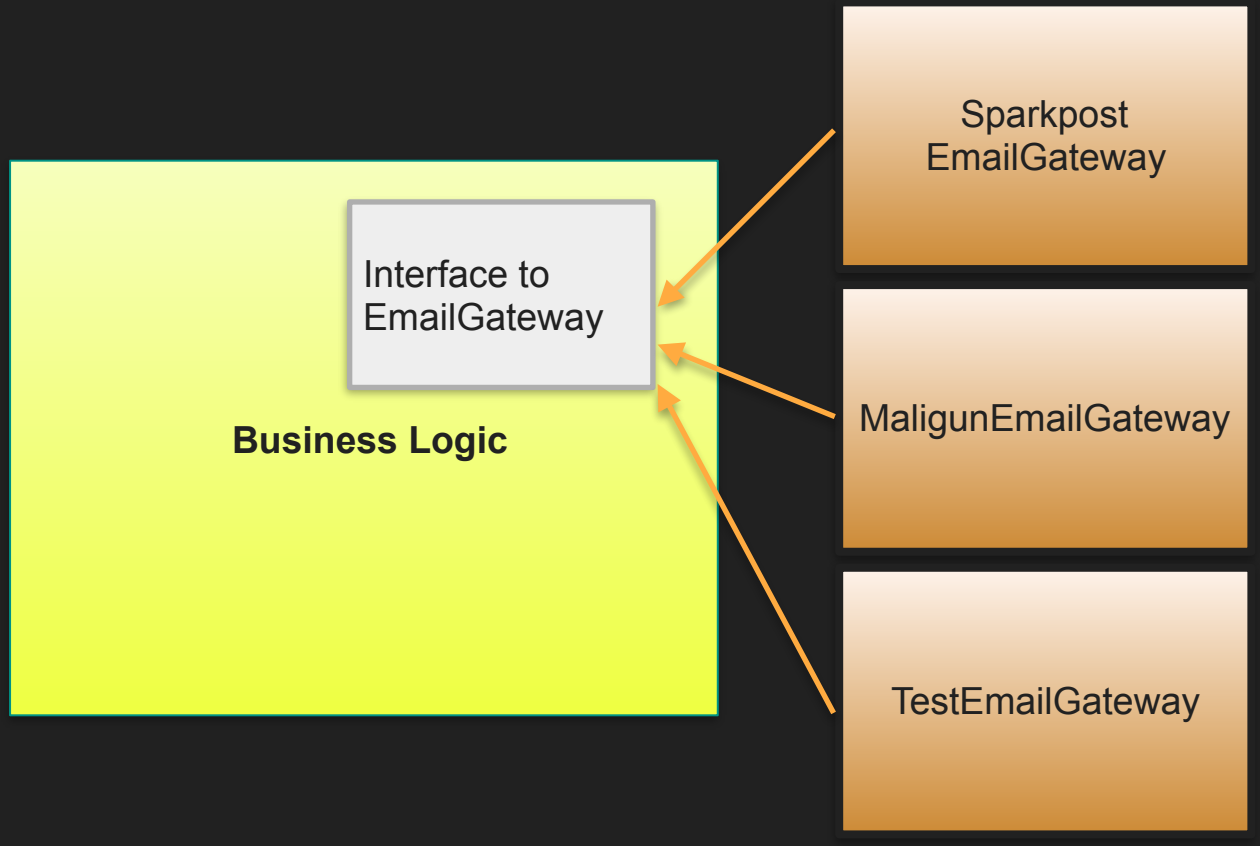
Existing tests - Correct

- Tests as before
- Stub for RecentPasswordChecker
- isRecentPassword always returns false

Take away:
Use stubs unless you actually need mocks

Architecture: Bigger tests


External Service
(e.g. EmailGateway)
Interface to
external service
Business Logic


Interface to EmailGateway
Business Logic
Sparkpost EmailGateway
MaligunEmailGateway
TestEmailGateway

Email Gateway Interface

```php
interface EmailGatewayInterface
{
    /**
     * Gateway for sending and email
     *
     * @param EmailMessage $message to send
     */
      public function sendEmail(EmailMessage $message);
}
```

EmailMessage

To

From

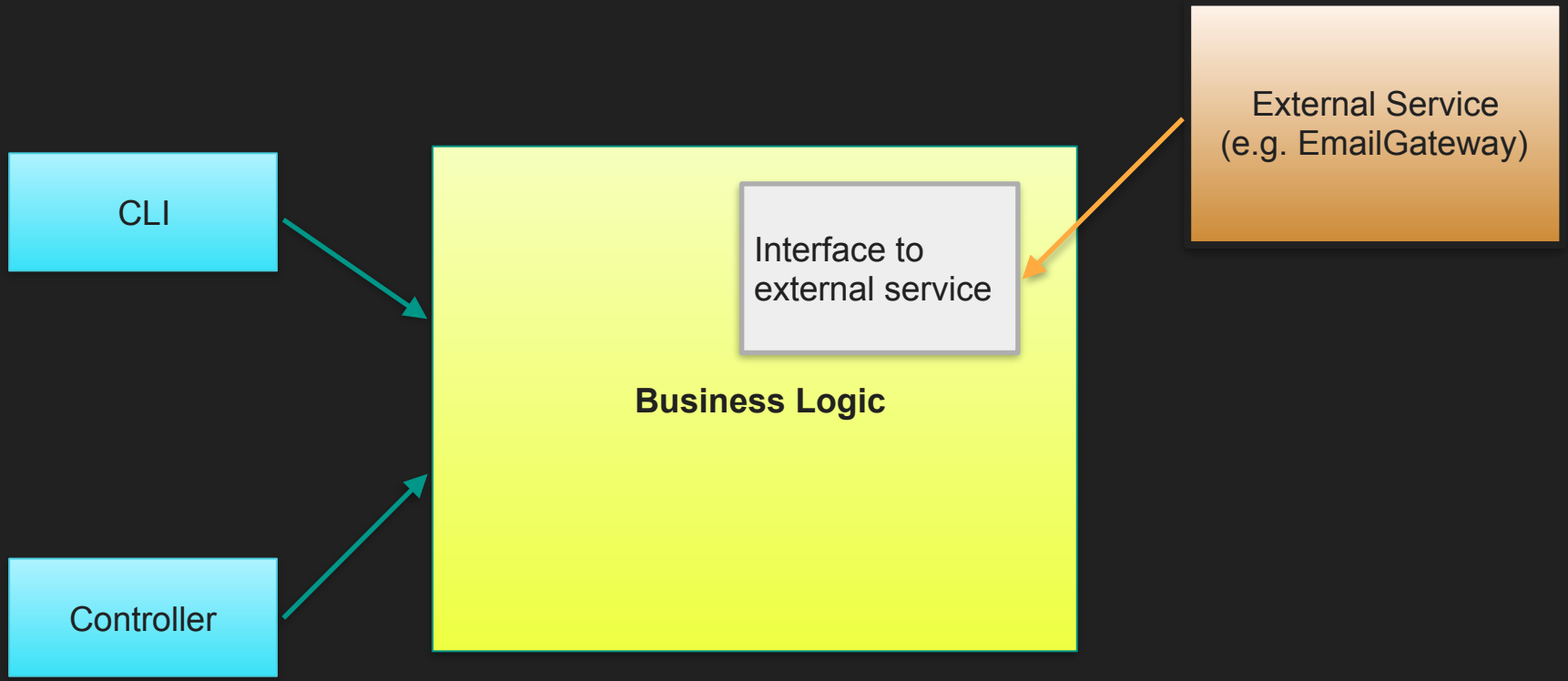
CC

Subject

Message Body

Template Name

Template Data

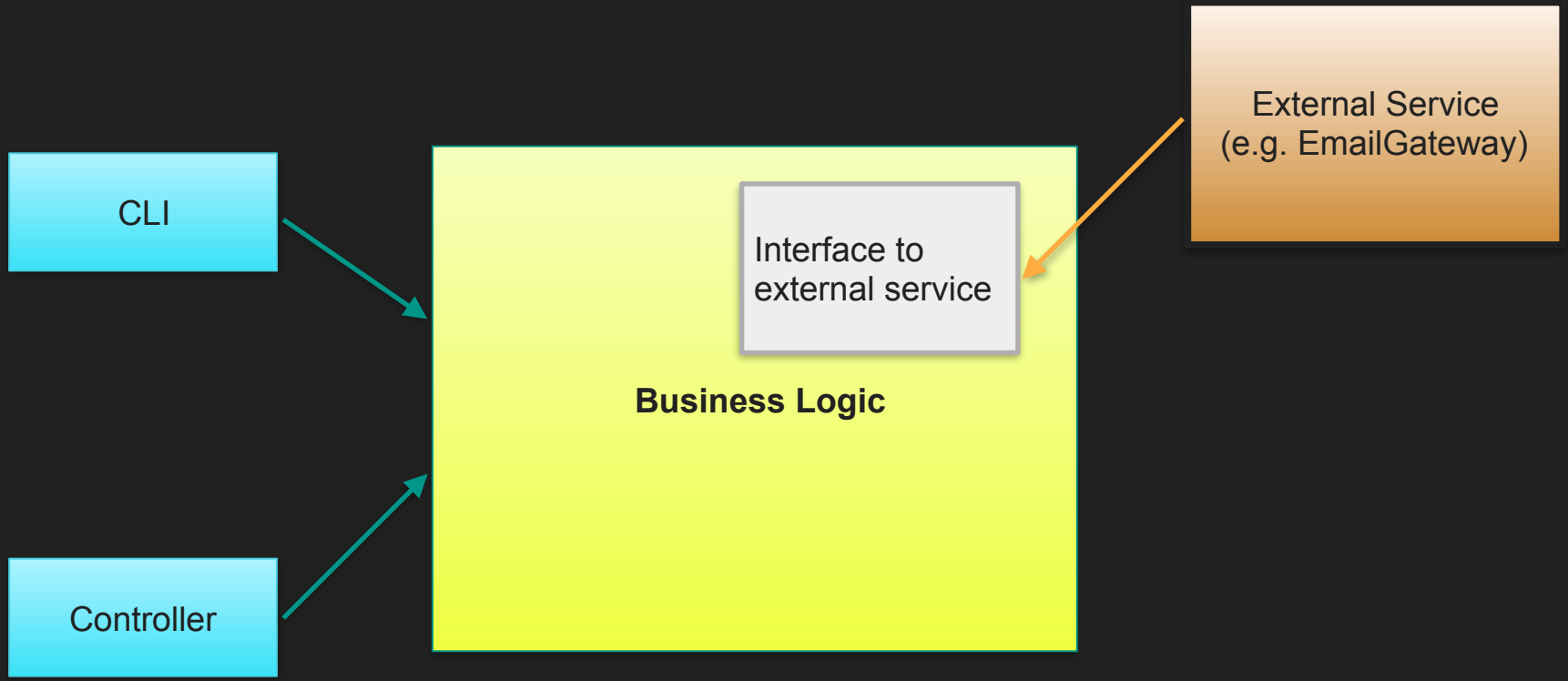

CLI
Controller
Interface to external service
Business Logic
External Service (e.g. EmailGateway)

Thin Controllers

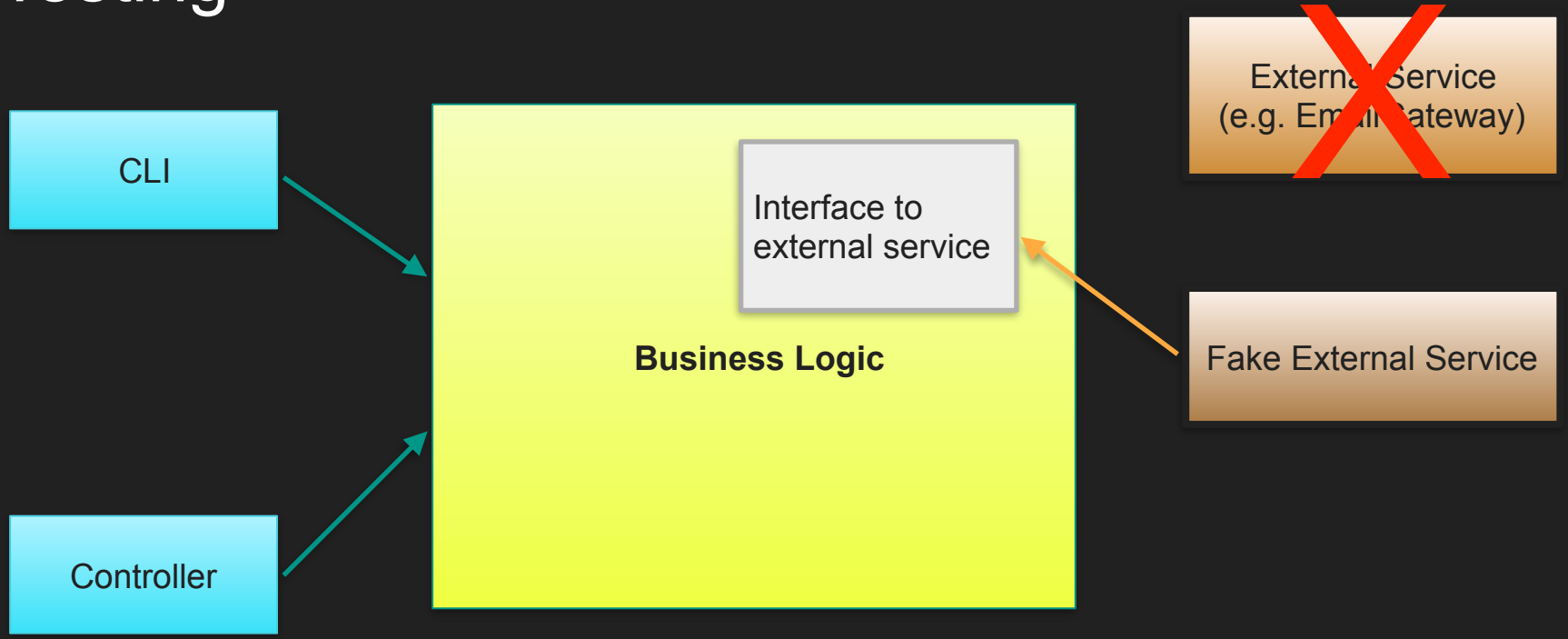
```
class UserController
{
  public function confirmUser()
  {
    $token = Input::get("token");
    $success = $this->userService->confirmUser($token);

    if ($success) {
      // Handle success
    } else {
      // Handle failure
    }
  }
```

Thin Controllers

```
class UserController
{
        public function confirmUser()
        {
        $token = Input::get("token");
          $success = $this->userService->confirmUser($token);

        if ($success) {
          // Handle success
        } else {
          // Handle failure
        }
      }
```


CLI
Controller
Interface to external service
Business Logic
External Service (e.g. EmailGateway)

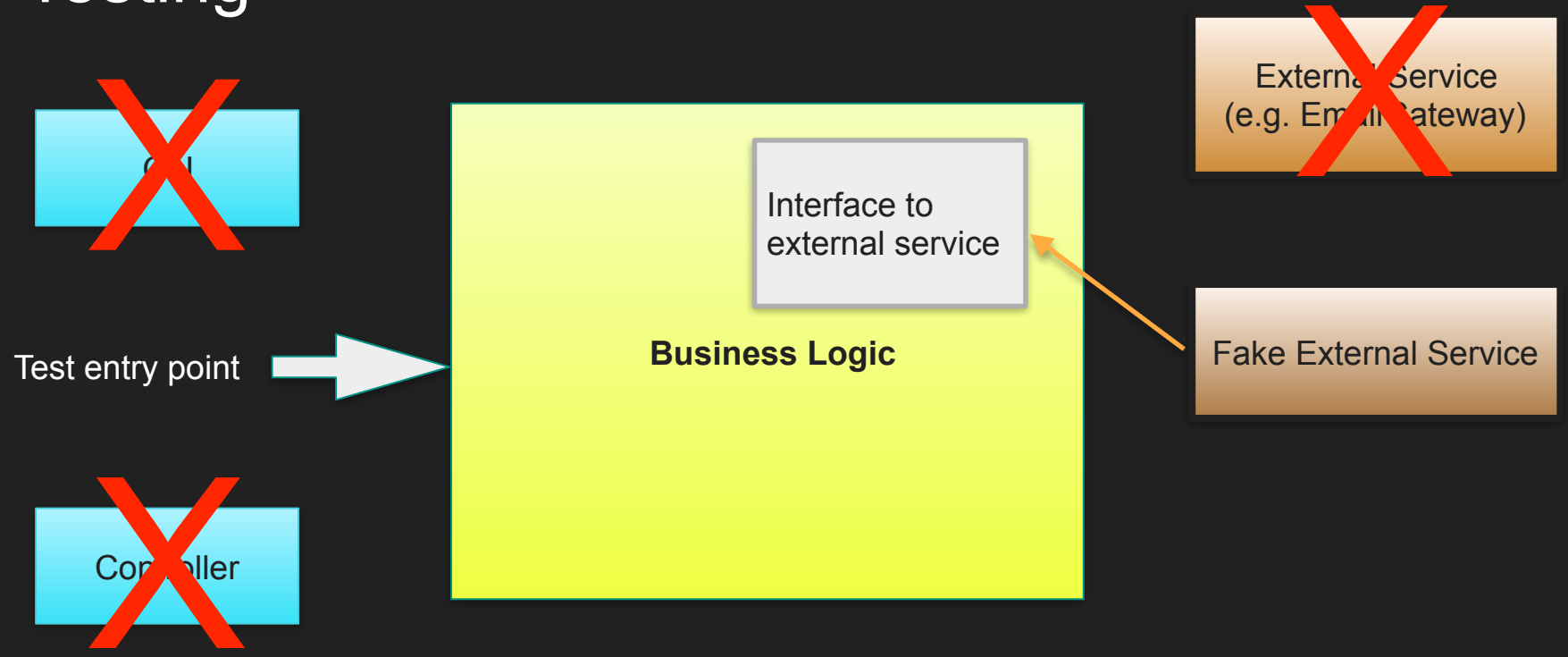
Testing


CLI
Controller
Interface to external service
Business Logic
Fake External Service

Email Gateway Fake

```
class EmailGatewayFake implements EmailGatewayInterface
{
    public function sendEmail(EmailMessage $message)
    {
      /* implementation that stores all messages for searching */
    }

    /**
     * Find emails that would have been sent
     *
     * @param array $criteria e.g.:
     *      ['to' => 'dave@example.com', 'template' => 'RegisterUser']
     * @return EmailMessage[] messages that meet criteria
     */
     public function findEmails(array $criteria)

}
```

Testing


Interface to external service
Business Logic
Test entry point
Fake External Service

Testing User Registration

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser(“dave@example.com”, “1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          [“to” => “dave@example.com”, “template” => “RegisterUser”]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data[“confirmationToken”];

        // Complete registration
        $this->assertTrue($userService->confirmUser($confirmationToken));
```

Testing User Registration

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayFake and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $this->assertTrue($userService->confirmUser($confirmationToken));
```

Testing User Registration

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $this->assertTrue($userService->confirmUser($confirmationToken));
```

Testing User Registration

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $this->assertTrue($userService->confirmUser($confirmationToken));
```

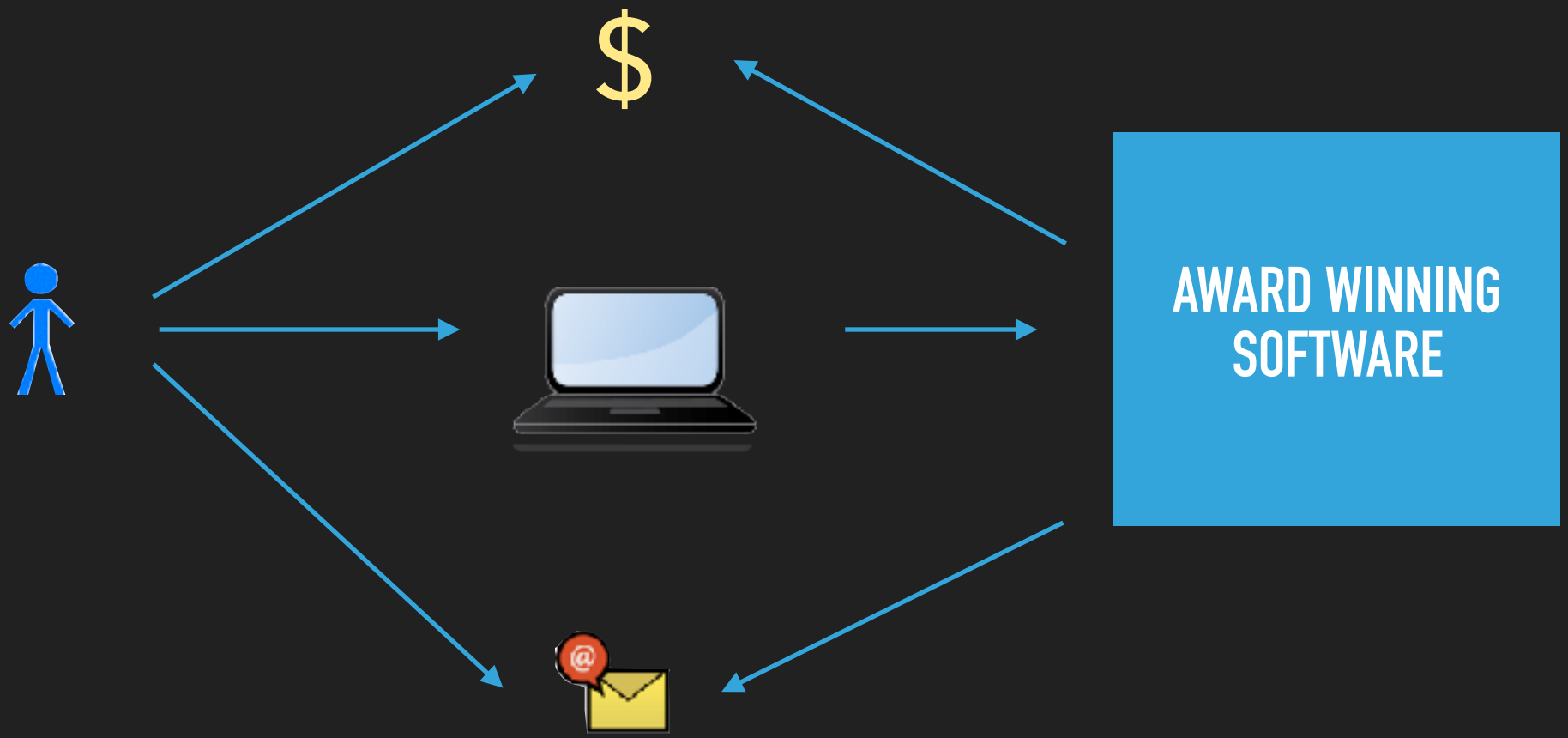
A codebase that's
easy to test
is probably
well architected

Can we automate anything else?

Automating as much as we can:

```
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]
```

Still need manual system tests



Summary

#1 We need a test suite

- Proves code works
- Stops regression
- Enables refactoring

#2 Ideal test suite…

- Fast to execute
- High coverage
- Low maintenance

#3 Write testable code

- Well architected
- Easy to maintain
- Easier to automate tests

Questions