Beat The Bugs Before They Beat You

Dave Liddament (@daveliddament)

Question 1:
Who puts bugs in their code?

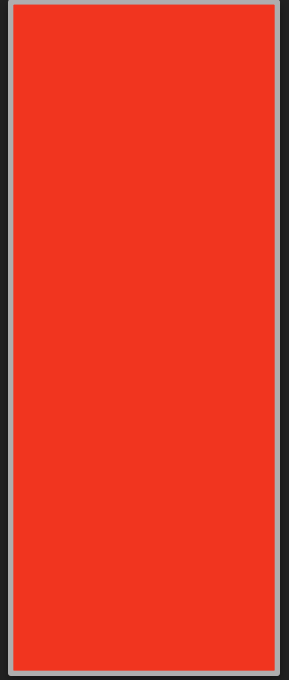
Question 2:
When is the best time to find a bug?

Best time to find a bug?

Best time to find a bug?

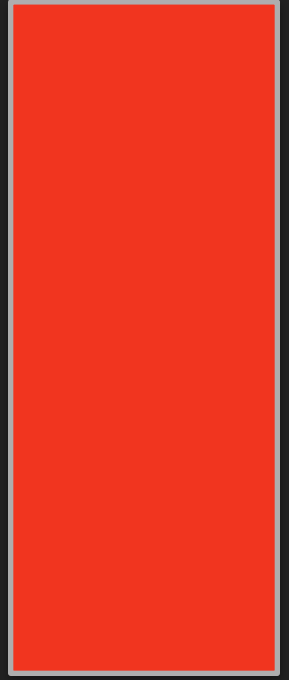
Months
into
operation

Best time to find a bug?



Months into operation

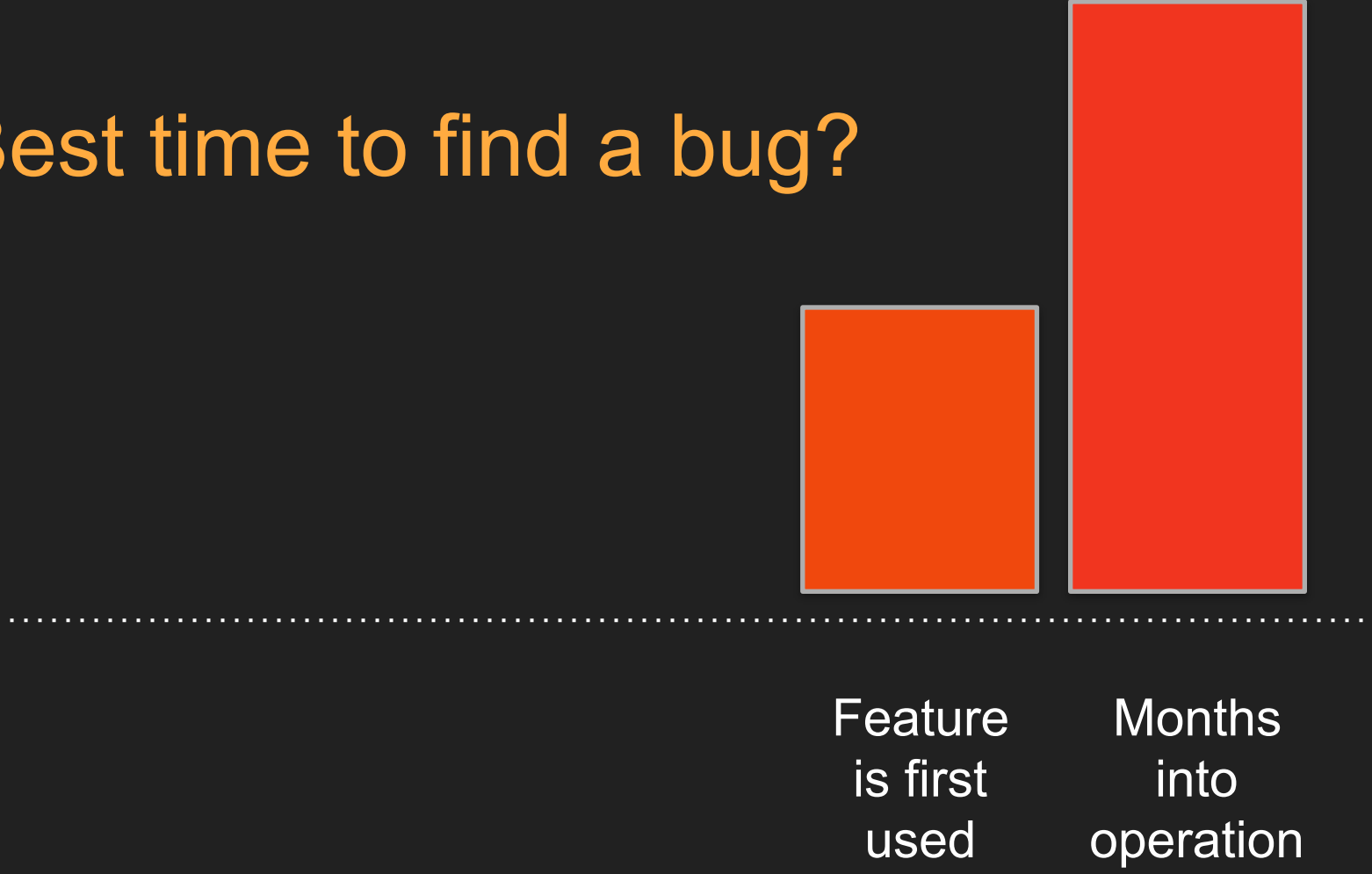
Best time to find a bug?



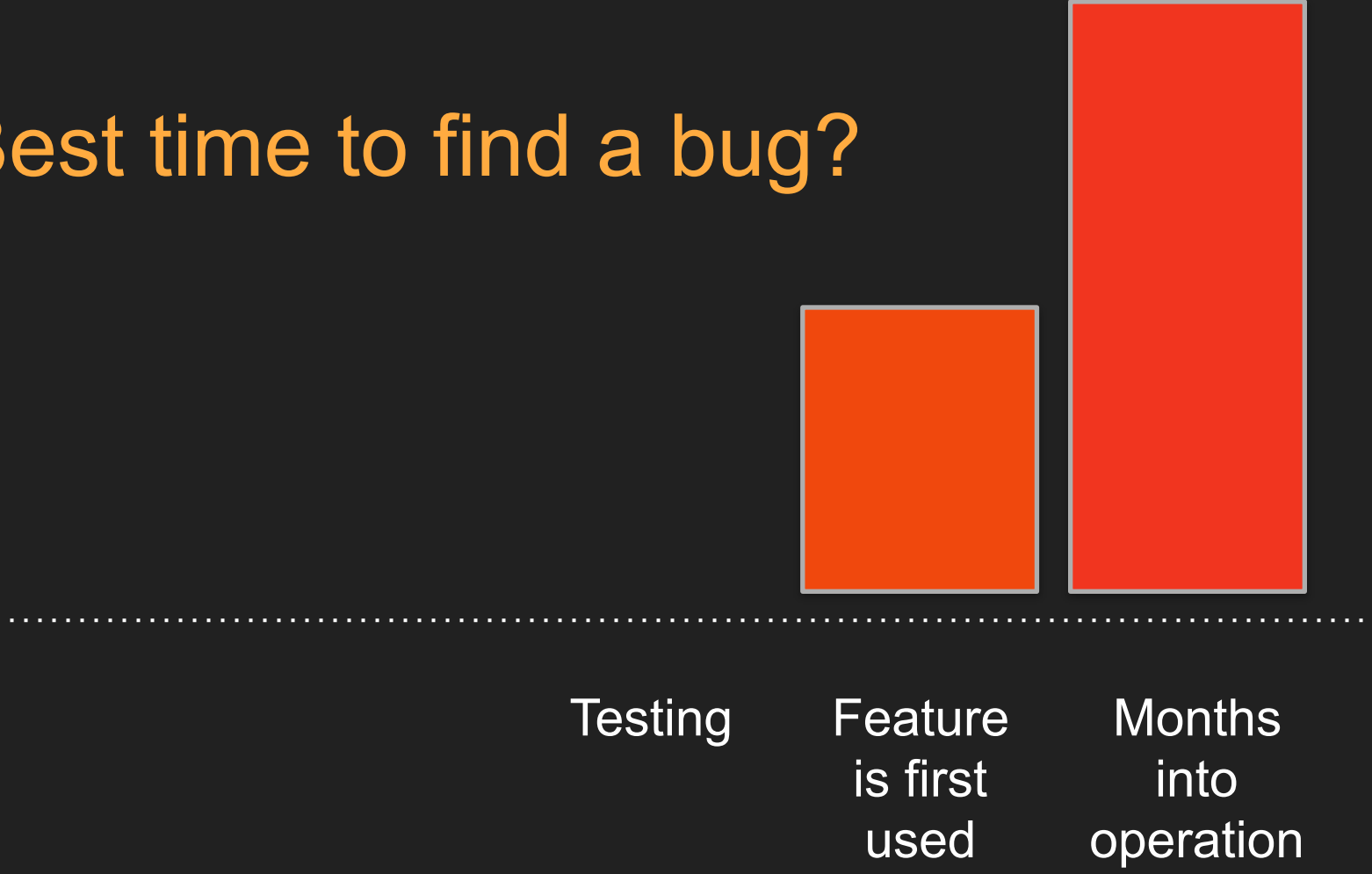
Feature is first used

Months into operation

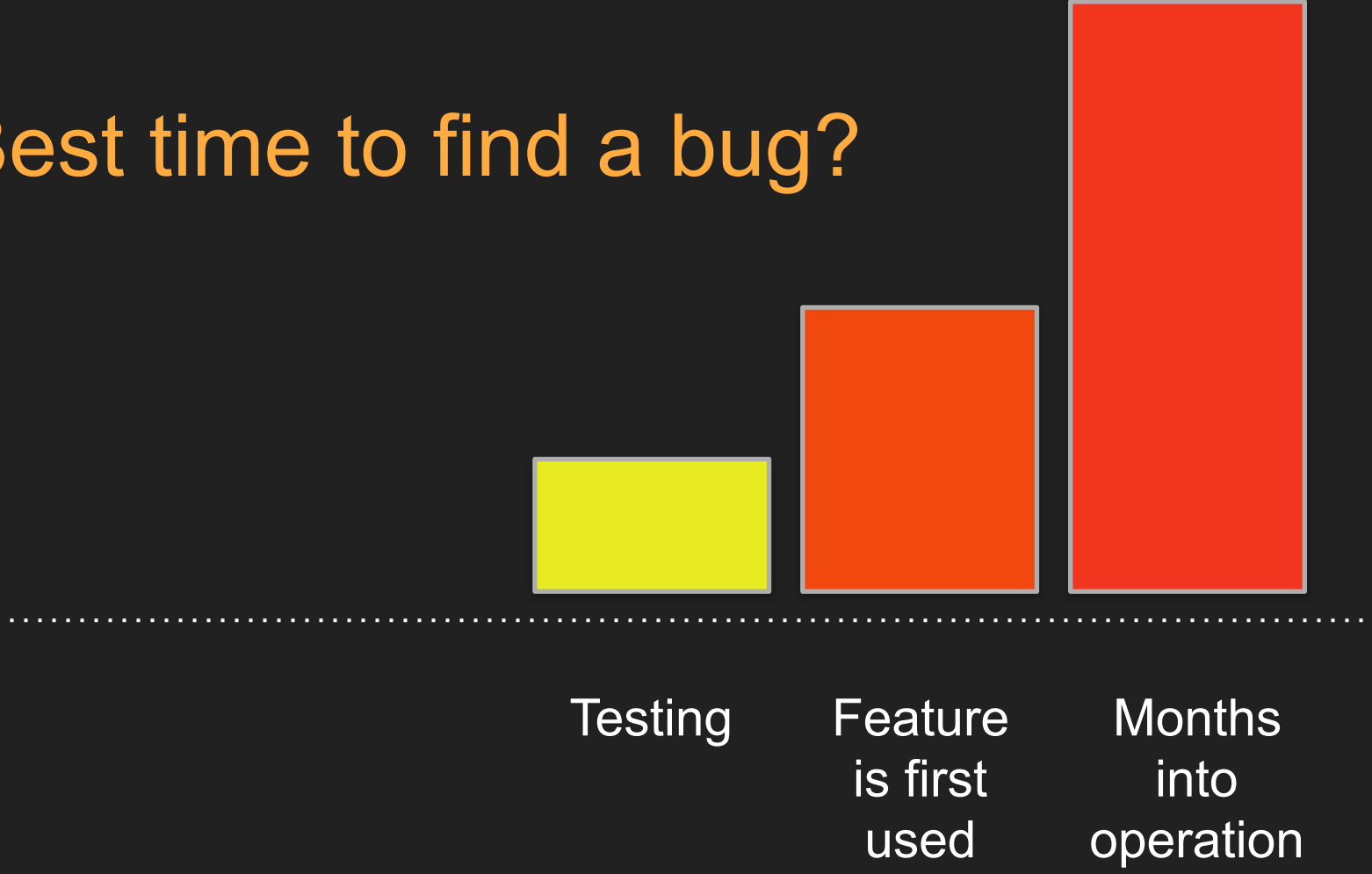
Best time to find a bug?



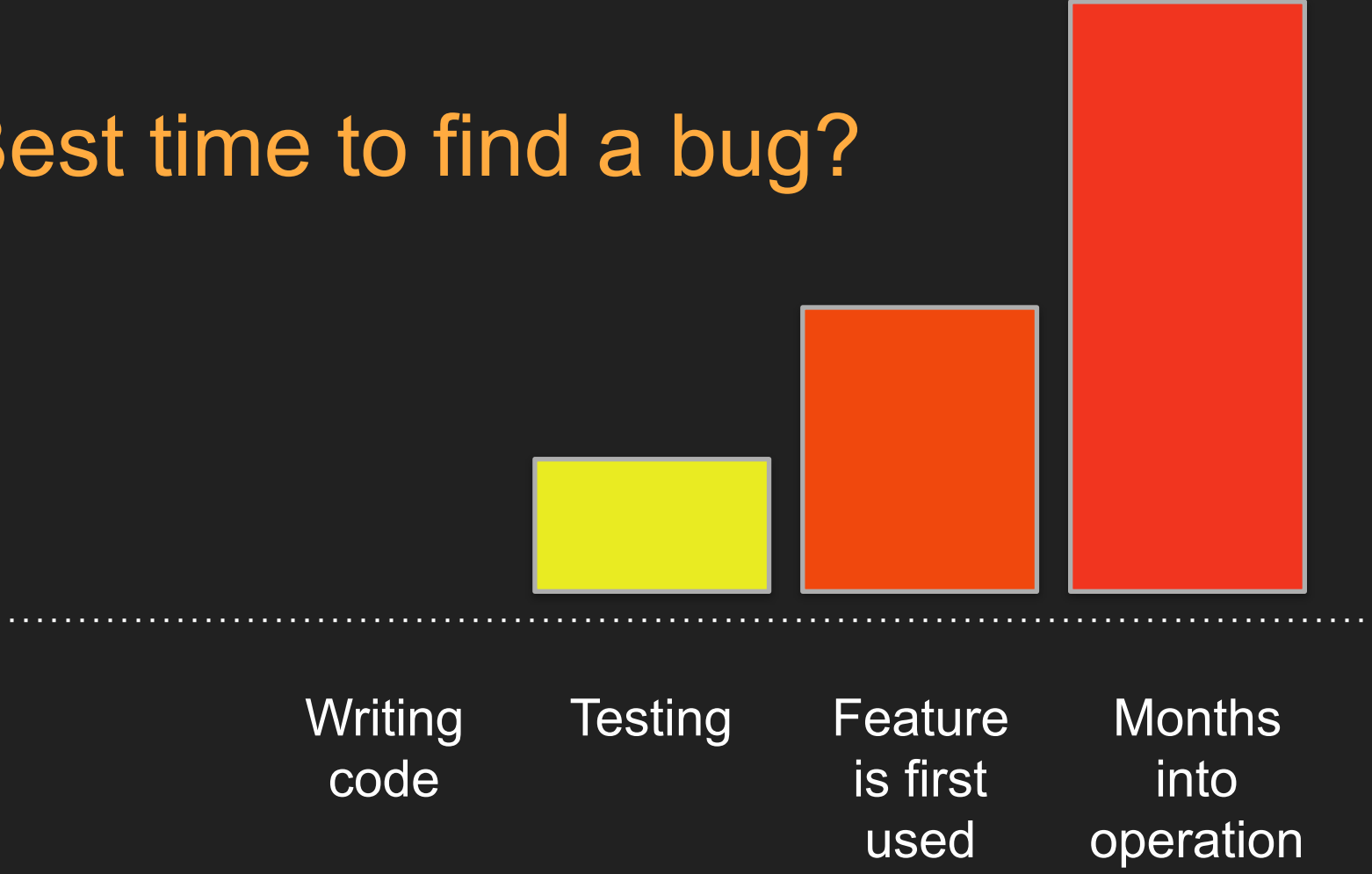
Best time to find a bug?



Best time to find a bug?



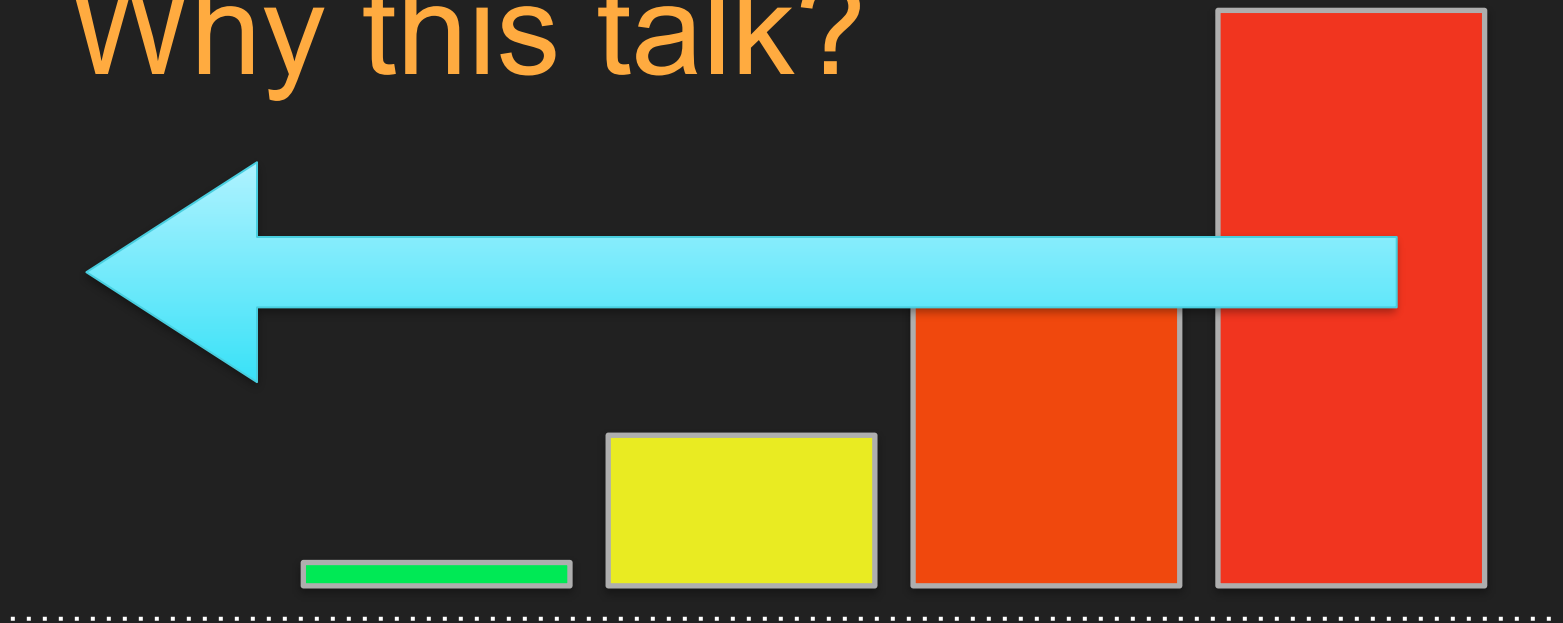
Best time to find a bug?



Best time to find a bug?

Best time to find a bug?

Why this talk?



Before writing code

Writing code

Testing

Feature is first used

Months into operation

Dave Liddament

@daveliddament

Lamp Bristol

Dave Liddament @daveliddament

Lamp Bristol

15+ years software development (PHP, Java, Python, C)

Dave Liddament @daveliddament

Lamp Bristol

15+ years software development (PHP, Java, Python, C)

Responsible for many thousands of bugs

Dave Liddament @daveliddament

Lamp Bristol

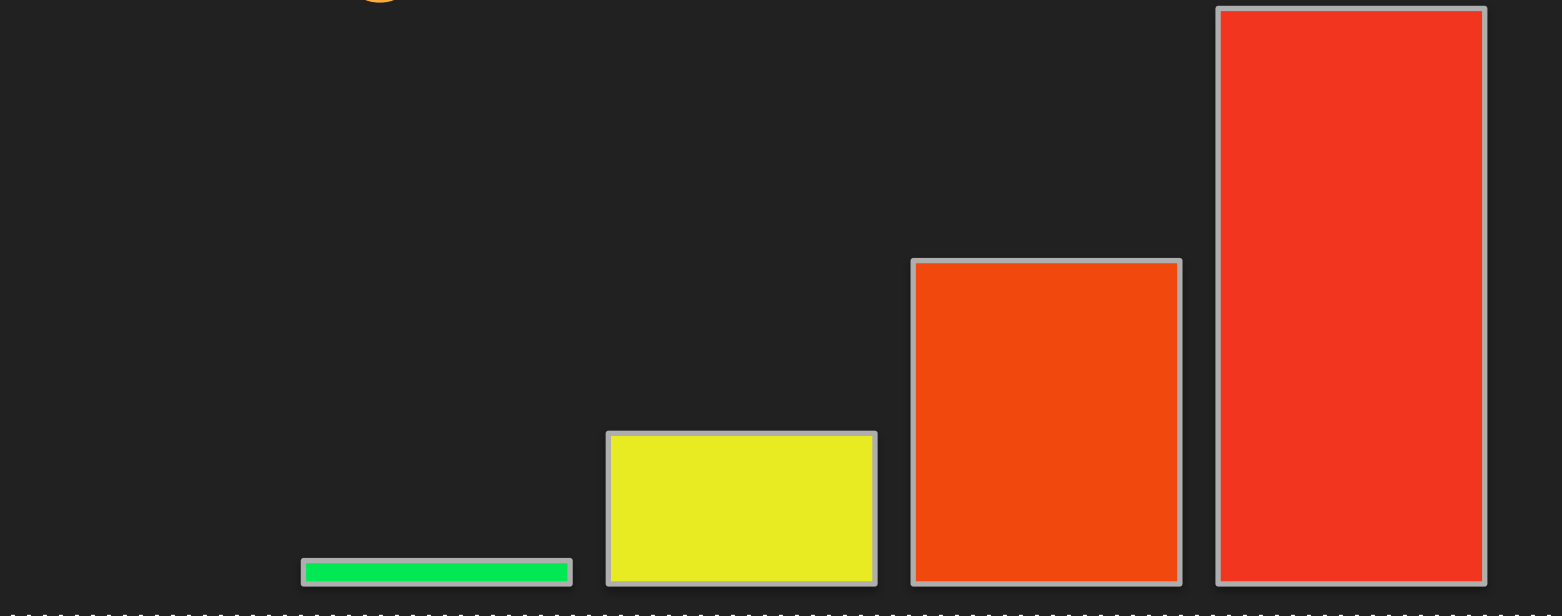
15+ years software development (PHP, Java, Python, C)

Responsible for many thousands of bugs

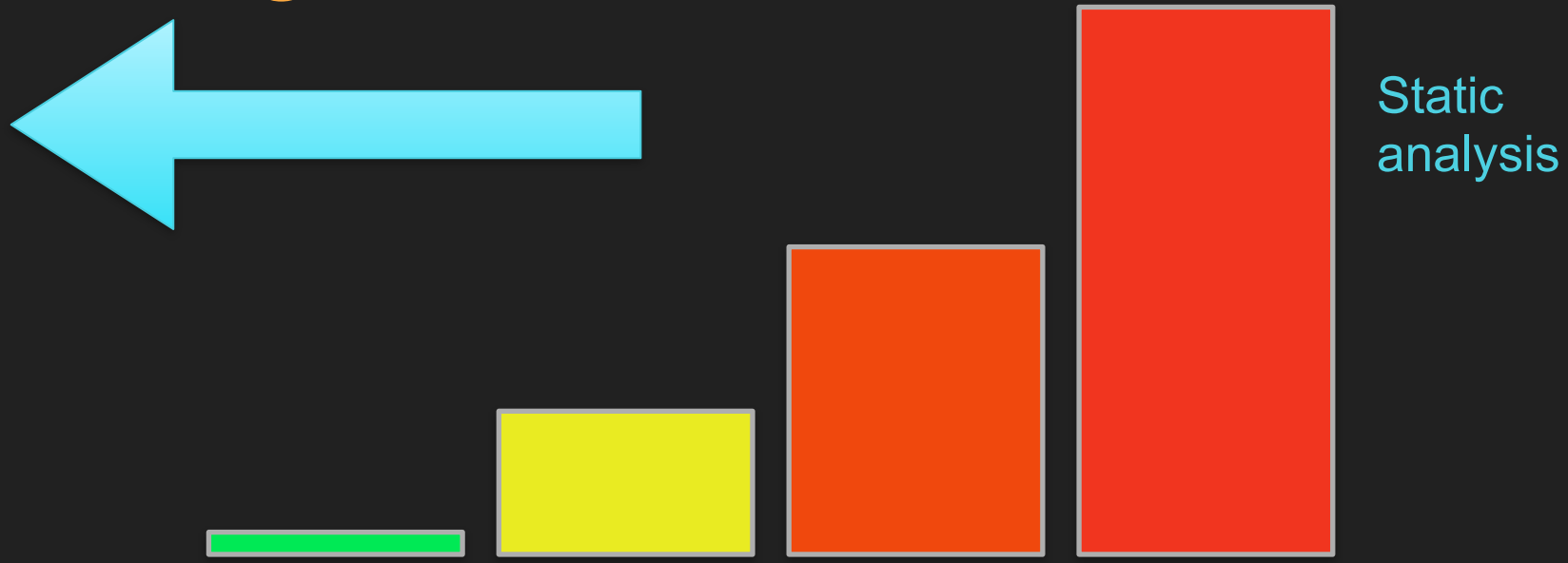
Organise PHP-SW user group and Bristol PHP Training

Agenda

Agenda


Before writing code
Writing code
Testing
Feature is first used
Months into operation

Agenda


Static analysis
Before writing code
Writing code
Testing
Feature is first used
Months into operation

Agenda

Before writing code

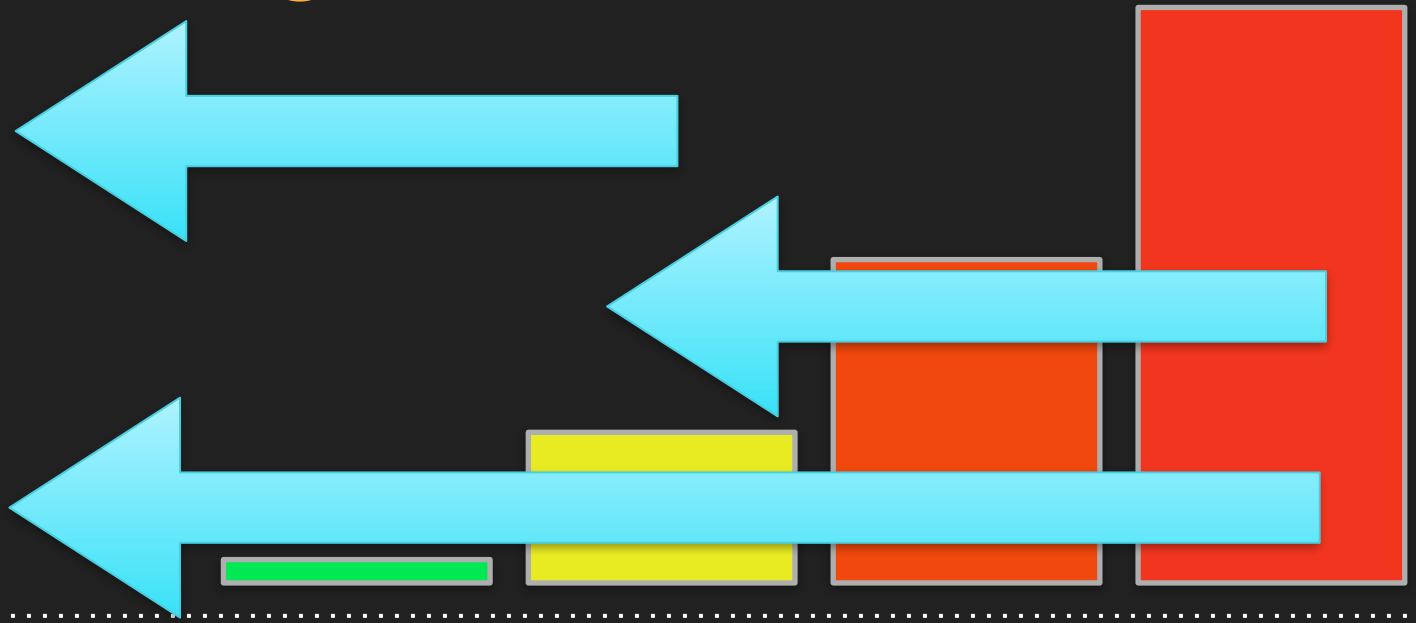
Writing code

Testing

Feature is first used

Months into operation

Agenda



Before writing code

Writing code

Testing

Feature is first used

Months into operation

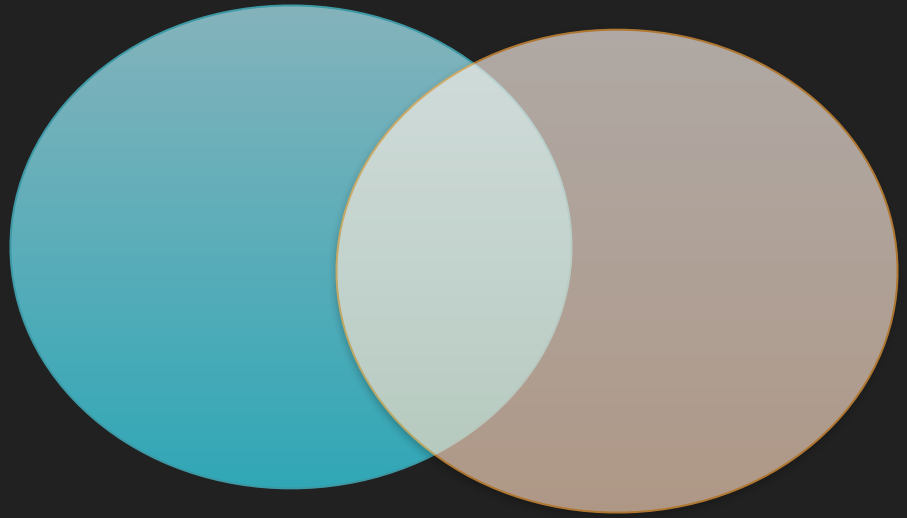
Why do bugs happen?

Why do bugs happen?

What the code should do

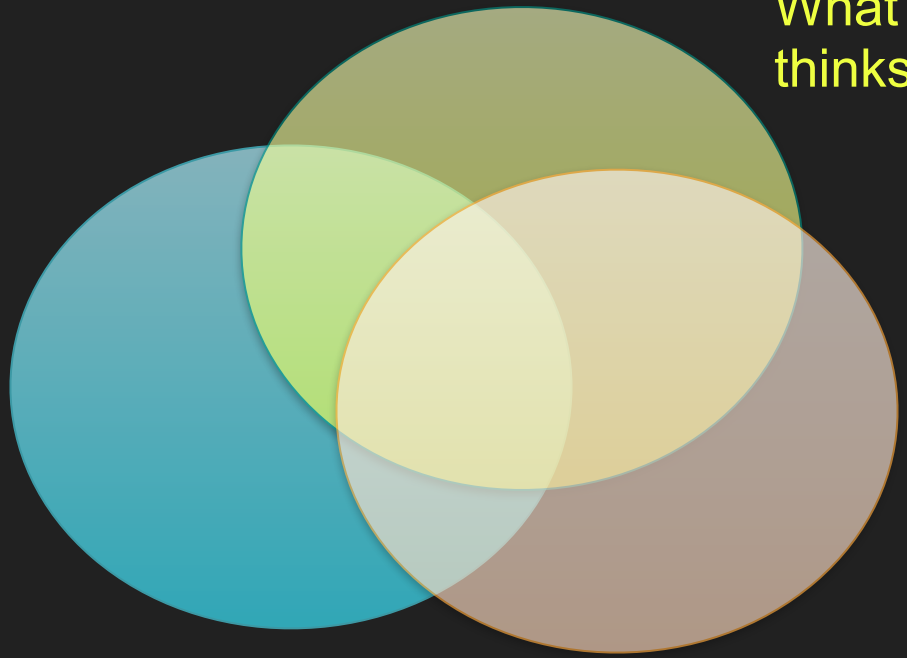
Why do bugs happen?

What the code should do

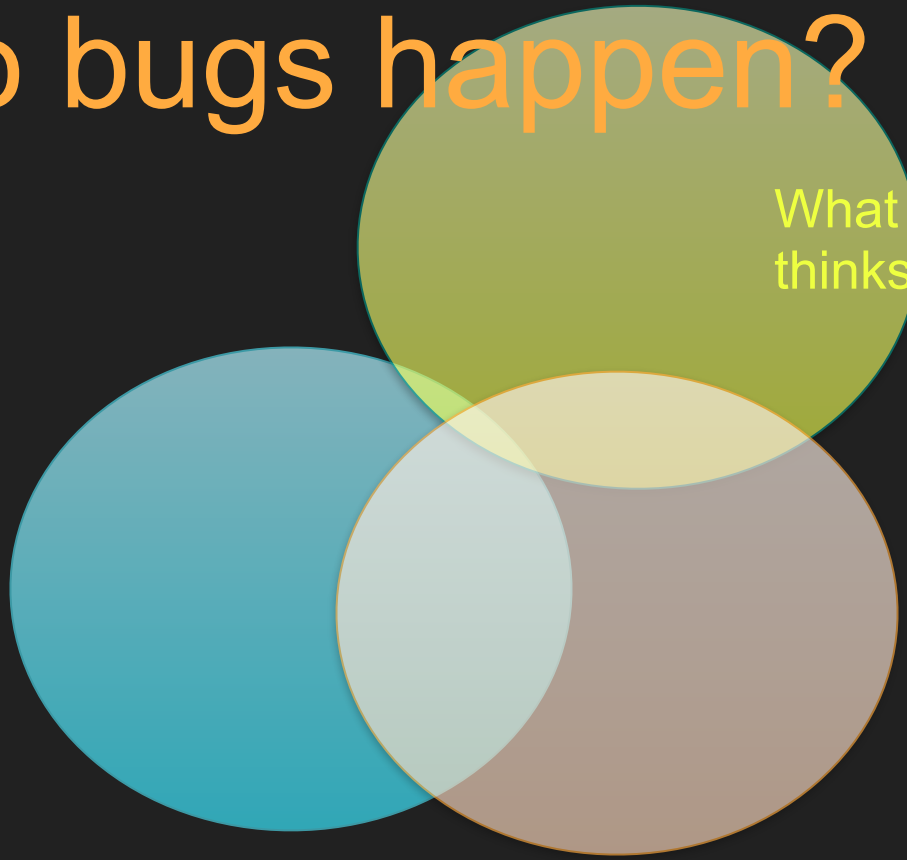


What the code actually does

Why do bugs happen?



Why do bugs happen?



Static analysis

Is this code valid?

```
function process($user) {
 // some implementation
}

$a = 1;
process($a);
```

Is this code valid?

```
function process($user) {
 // some implementation
}

$a = 1;
process($a);
```

Is this code valid?

```
function process($user) {
 // some implementation
}

$a = 1;
process($a);
```

Is this code valid?

```
function process($user) {
 // some implementation
}

$a = 1;
process($a);
```

Is this code valid?

```
function process(User $user) {
 // some implementation
}

$a = 1;
process($a);
```

Is this code valid?

```
function process(User $user) {
 // some implementation
}

$a = 1;
process($a);
```

Is this code valid?

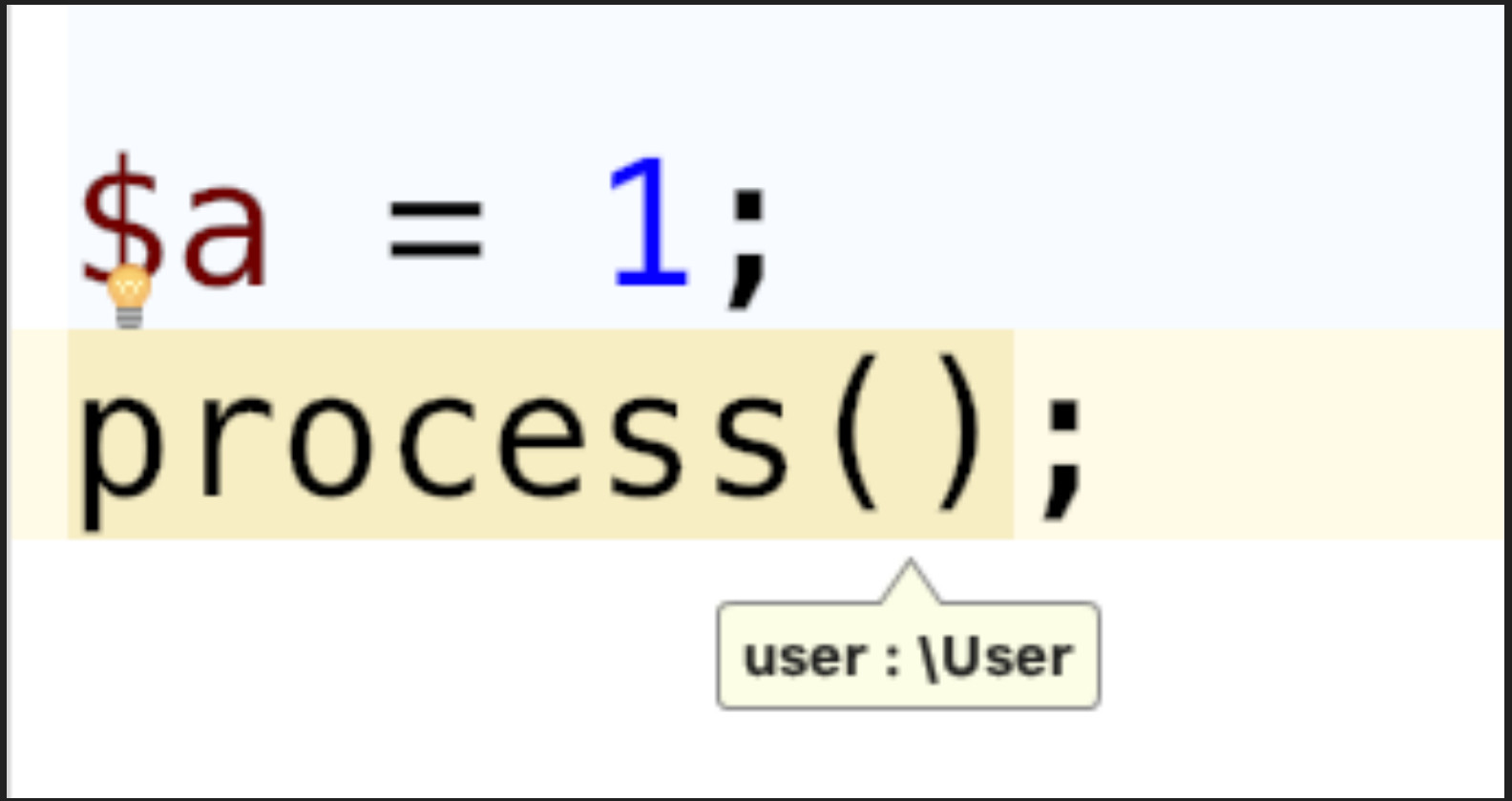
```
function process(User $user) {
 // some implementation
}

$a = 1;
process($a);
```

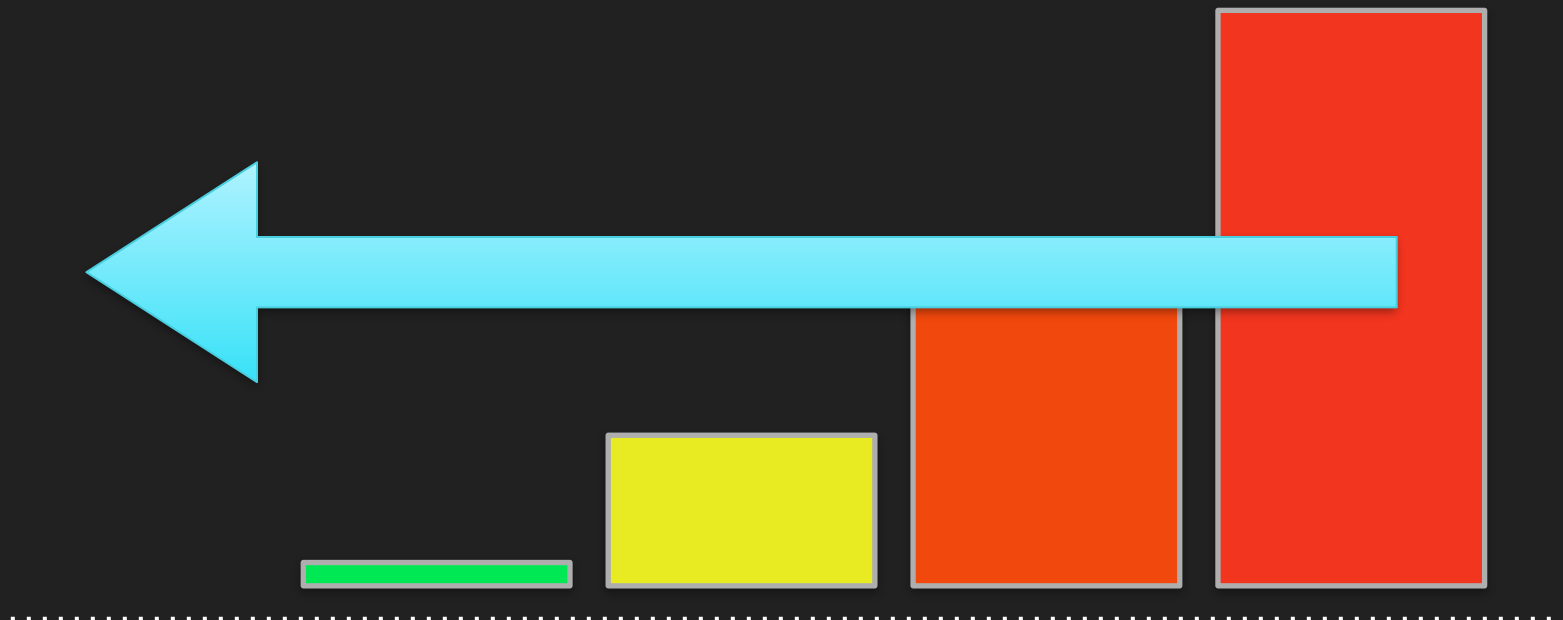
Is this code valid?

```
function process(User $user) {
 // some implementation
}

$a = 1;
process($a);
```

```php
function process(User $user) {
    // some implementation
}

$a = 1;
process($a);
```

Expected User, got int more... (⌘F1)


$a = 1;
process();
user : \User



Before writing code

Writing code

Testing

Feature is first used

Months into operation

Type hinting has helped

```
function process(User $user) {
 // some implementation
}

$a = 1;
process($a);
```

Take away

Be explicit:
Type hint everything

Take away

Use a modern IDE

More type hinting with PHP 7

```
function duplicateString (
    string $value,
    int $times) :string
```

More type hinting with PHP 7

```
function duplicateString (
    string $value,
    int $times) :string
```

More type hinting with PHP 7

```
function duplicateString (
    string $value,
    int $times) :string
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Is this code valid?

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(12);
process($a);
```

Language level validation

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(“dave”);
process($a);
```

Language level validation

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(“dave”);
process($a);
```

Language level validation

```
function getUser(int $id): User {…}

function process(User $user): void {…}

$a = getUser(“dave”);
process($a);
```

Gap in PHP type system: Generics

```
function getUsers(): array
{
    … get $user1, $user2, $user3 …

    return [$user1, $user2, $user3];
}
```

https://wiki.php.net/rfc/generics

Gap in PHP type system: Generics

```
function getUsers(): array
{
    … get $user1, $user2, $user3 …

    return [$user1, $user2, $user3];
}
```

https://wiki.php.net/rfc/generics

Gap in PHP type system: Generics

```
function getUsers(): array
{
    … get $user1, $user2, $user3 …

    return [$user1, $user2, $user3];
}
```

https://wiki.php.net/rfc/generics

Gap in PHP type system: Generics

```
function getUsers(): array<User>
{
    … get $user1, $user2, $user3 …

    return [$user1, $user2, $user3];
}
```

https://wiki.php.net/rfc/generics

A very important PHP contribution…

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}
```

```
/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

PHPStorm can simulate limited generics

```
class User {
  public function getAccountNumber() :string {…}
}
```

```
/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getAccountNumber();
}
```

Static analysis can find errors

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getSomething();
}
```

Static analysis can find errors

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getSomething();
}
```

Static analysis can find errors

```
class User {
  public function getAccountNumber() :string {…}
}

/**
 * @return User[]
 */
function getUsers(): array { … }

$users = getUsers();
foreach($users as $user) {
  $accountNumber = $user->getSomething();
}
```

Static analysis helps developers

More Type Hinting Hints

```
class User {
  public function deactivate(): void {…}
}

function deactivateUsers(iterable $users): void {
  /**
   * @var string $name
   * @var User $user
   */
  foreach($users as $name => $user) {
    echo “Deactivating [$name]”;
    $user->deactivate();
  }
}
```

More Type Hinting Hints

```
class User {
  public function deactivate(): void {…}
}
```

```
function deactivateUsers(iterable $users): void {
  /**
   * @var string $name
   * @var User $user
   */
  foreach($users as $name => $user) {
    echo “Deactivating [$name]”;
    $user->deactivate();
  }
}
```

More Type Hinting Hints

```
class User {
  public function deactivate(): void {…}
}

function deactivateUsers(iterable $users): void {
  /**
   * @var string $name
   * @var User $user
   */
  foreach($users as $name => $user) {
    echo “Deactivating [$name]”;
    $user->deactivate();
  }
}
```

More Type Hinting Hints

```
class User {
  public function deactivate(): void {…}
}

function deactivateUsers(iterable $users): void {
  /**
   * @var string $name
   * @var User $user
   */
  foreach($users as $name => $user) {
    echo “Deactivating [$name]”;
    $user->deactivate();
  }
}
```

More Type Hinting Hints

```
class User {
  public function deactivate(): void {…}
}

function deactivateUsers(iterable $users): void {
  /**
   * @var string $name
   * @var User $user
   */
  foreach($users as $name => $user) {
    echo “Deactivating [$name]”;
    $user->deactivate();
  }
}
```

More Type Hinting Hints

```
class User {
  public function deactivate(): void {…}
}

function deactivateUsers(iterable $users): void {
  /**
   * @var string $name
   * @var User $user
   */
  foreach($users as $name => $user) {
    echo “Deactivating [$name]”;
    $user->deactivate();
  }
}
```

Take away

Be explicit:
Use Docblock type hints for generics

Use void

```
function deactivate(): void {…}

$foo = deactivate();
```

Use void

```
function deactivate(): void {…}

$foo = deactivate();
```

Use void

```
function deactivate(): void {…}

$foo = deactivate();
```

Use void

```
function deactivate(): void {…}

$foo = deactivate();
```

Static analysis recap

Static analysis recap

- Analyse code without running it

Static analysis recap

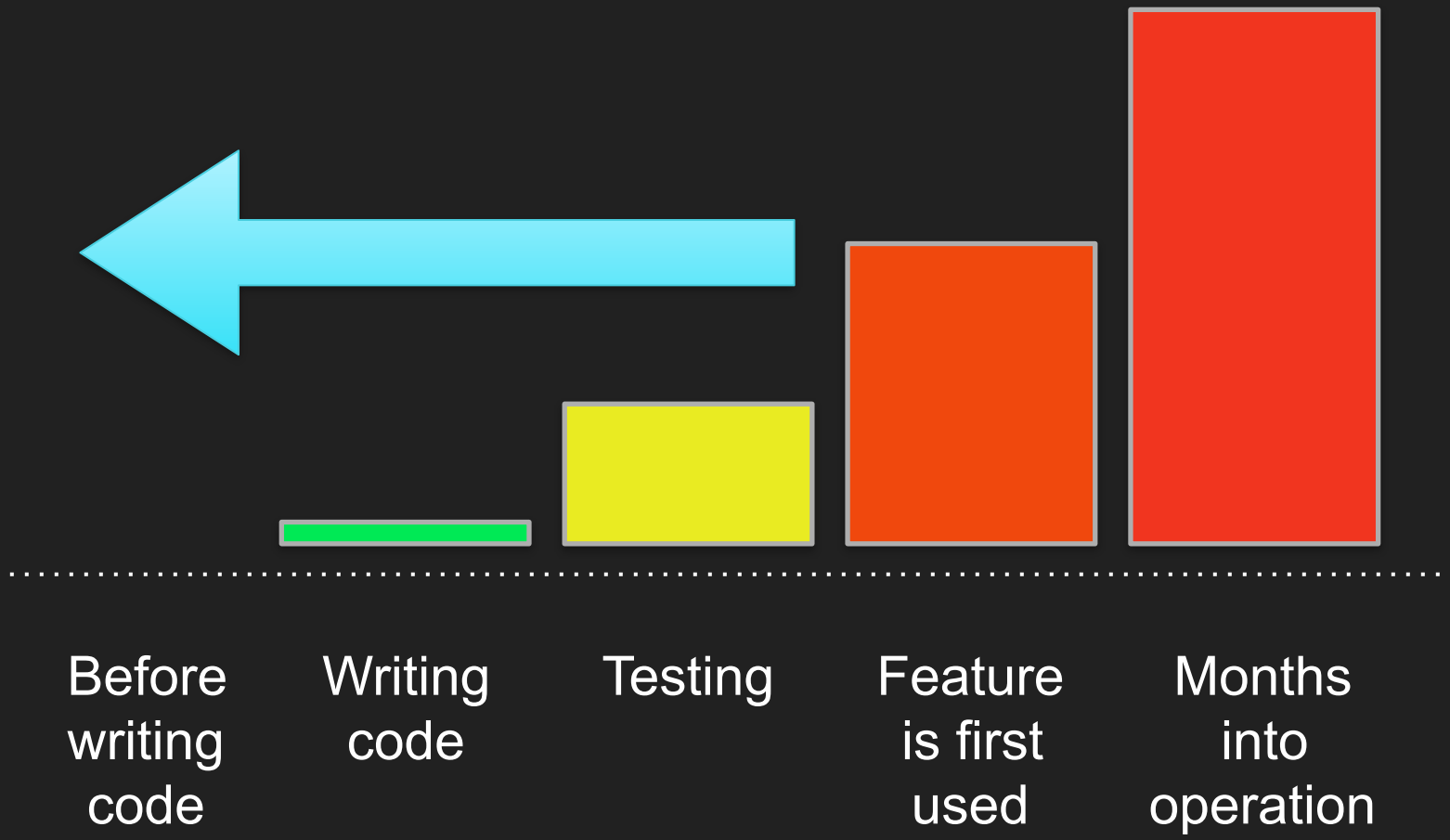
- Analyse code without running it
- Prevent bugs even entering the code base

Static analysis recap

- Analyse code without running it
- Prevent bugs even entering the code base
- Type hinting and doc blocks comments help static analysis tools
    - which in turn help developers

Static analysis recap

- Analyse code without running it
- Prevent bugs even entering the code base
- Type hinting and doc blocks comments help static analysis tools
    - which in turn help developers
- Use an IDE that offers static analysis


Before writing code
Writing code
Testing
Feature is first used
Months into operation

Static analysis is no silver bullet

Run time analysis

Run time analysis

- Testing

Run time analysis

- Testing
- Assertions

Run time analysis: Testing

Bugs will only be found in code that is executed

```
function foo(bool $bar): void
{
    if ($bar) {
        … do something …
    } else {
        … code with a bug …
    }
}
```

Bugs will only be found in code that is executed

```
function foo(bool $bar): void
{
    if ($bar) {
        … do something …
    } else {
        … code with a bug …
    }
}
```

Bugs will only be found in code that is executed

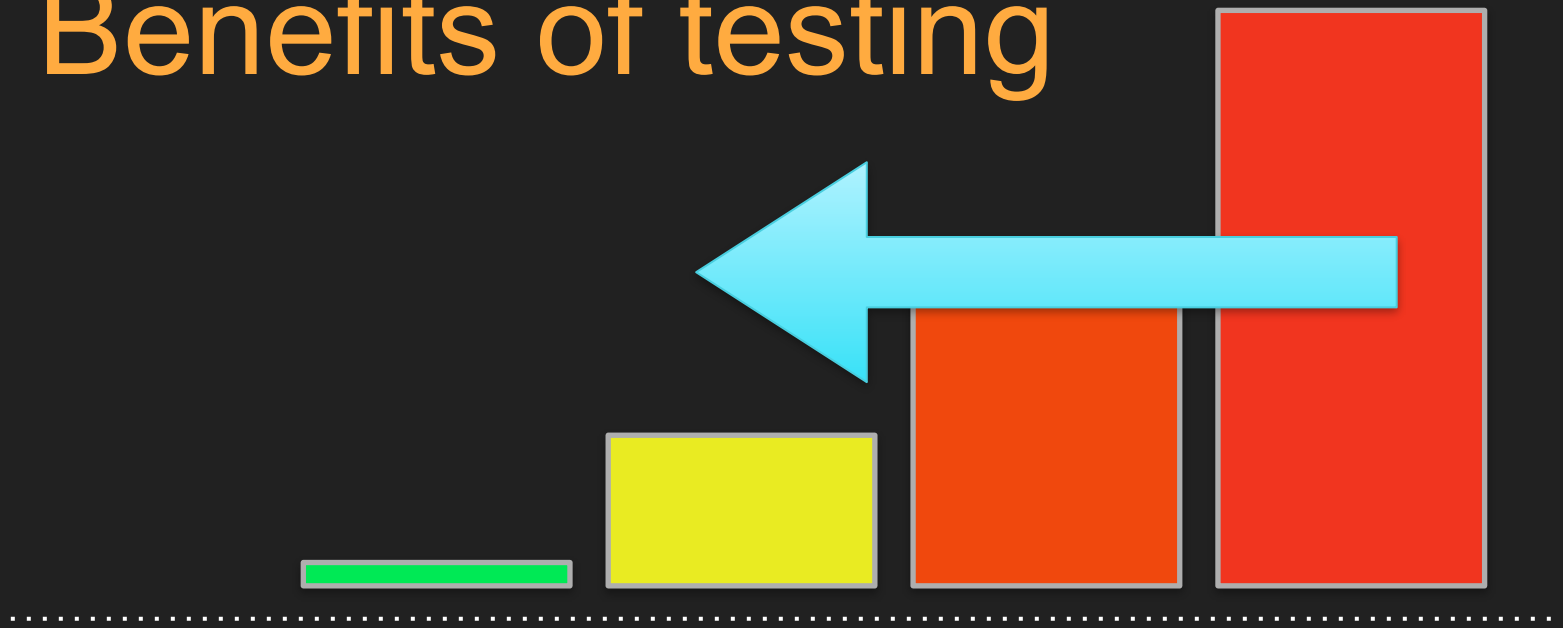
```
function foo(bool $bar): void
{
    if ($bar) {
        … do something …
    } else {
        … code with a bug …
    }
}
```

Bugs will only be found in code that is executed

```
function foo(bool $bar): void
{
    if ($bar) {
        … do something …
    } else {
        … code with a bug …
    }
}
```

Higher our ‘code coverage’
the more bugs we’ll find

Benefits of testing



Before writing code

Writing code

Testing

Feature is first used

Months into operation

More run time analysis: Assertions

More run time analysis: Assertions

Statements that the developer believes should always be true

Can we improve this code

```
public function setStatus(string $status){
    $this->status = $status;
}
```

Improvement 1: Add constants

```
const REGISTERED = ‘registered’;
const STARTED = ‘started’;
const FINISHED = ‘finished’;
const QUIT = ‘quit’;

public function setStatus(string $status){
    $this->status = $status;
}
```

Improvement 1: Add constants

```
const REGISTERED = ‘registered’;
const STARTED = ‘started’;
const FINISHED = ‘finished’;
const QUIT = ‘quit’;
```

```
public function setStatus(string $status){
    $this->status = $status;
}
```

Improvement 2: Add assertion

```
… constants defined as before …

public function setStatus(string $status){
  if (!in_array($status,[self::REGISTERED,
        self::STARTED, self::FINISHED]) {
    throw new Exception(“Invalid status”);
  }
  $this->status = $status;
}
```

Improvement 2: Add assertion

```
… constants defined as before …

public function setStatus(string $status){
  if (!in_array($status,[self::REGISTERED,
        self::STARTED, self::FINISHED]) {
    throw new Exception(“Invalid status”);
  }
  $this->status = $status;
}
```

Improvement 2: Add assertion

```
… constants defined as before …

public function setStatus(string $status){
  if (!in_array($status,[self::REGISTERED,
        self::STARTED, self::FINISHED]) {
    throw new Exception("Invalid status");
  }
  $this->status = $status;
}
```

Create Assert class

```
class Assert {

public static function oneOf(
  $value,
  array $validValues,
  string $error) {

  if (!in_array($value, $validValues) {
    throw new Exception($error);
  }
}
```

Create Assert class

```
class Assert {

  public static function oneOf(
    $value,
    array $validValues,
    string $error) {

    if (!in_array($value, $validValues) {
      throw new Exception($error);
    }
  }
```

Create Assert class

```
class Assert {

public static function oneOf(
   $value,
   array $validValues,
   string $error) {

   if (!in_array($value, $validValues) {
      throw new Exception($error);
   }
}
```

Improvement 3: Use Assert class

```
… constants defined as before …

public function setStatus(string $status){
  Assert::oneOf(
   $status,
   self::REGISTERED, self::STARTED, self::FINISHED],
   “Invalid status”);

  $this->status = $status;
}
```

Improvement 3: Use Assert class

```
… constants defined as before …

public function setStatus(string $status){
  Assert::oneOf(
   $status,
   self::REGISTERED, self::STARTED, self::FINISHED],
   “Invalid status”);

  $this->status = $status;
}
```

Asserts

```
Assert::null
Assert::notNull
Assert::isEmpty
Assert::notEmpty
Assert::greaterThan
Assert::lessThan
…
```

Assertion Packages

Write your own

webmozart/assert

beberlei/assert

Specify a contract

```
/**
 * Returns Roman Numeral of $number.
 * NOTE: $number must be between 1 and 5000
 */
function asRomanNumeral(int $number): string {
  Assert::inRange($number, 1, 5000);

  … some implementation …
}
```

Specify a contract

```
/**
 * Returns Roman Numeral of $number.
 * NOTE: $number must be between 1 and 5000
 */
function asRomanNumeral(int $number): string {
  Assert::inRange($number, 1, 5000);

  … some implementation …
}
```

Specify a contract

```
/**
 * Returns Roman Numeral of $number.
 * NOTE: $number must be between 1 and 5000
 */
function asRomanNumeral(int $number): string {
  Assert::inRange($number, 1, 5000);

  … some implementation …
}
```

Code that worries me…

```
if ($type == 1) {
    $message = ‘hello’;
} elseif ($type == 2) {
    $message = ‘goodbye’;
}


sendMessage($message);
```

Code that worries me…

```
if ($type == 1) {
    $message = ‘hello’;
} elseif ($type == 2) {
    $message = ‘goodbye’;
}


sendMessage($message);
```

Code that worries me…

```
if ($type == 1) {
    $message = ‘hello’;
} elseif ($type == 2) {
    $message = ‘goodbye’;
}


sendMessage($message);
```

Code that worries me…

```
if ($type == 1) {
    $message = ‘hello’;
} elseif ($type == 2) {
    $message = ‘goodbye’;
}
```

```
sendMessage($message);
```

Now I’m happier…

```
if ($type == 1) {
    $message = ‘hello’;
} elseif ($type == 2) {
    $message = ‘goodbye’;
} else {
    throw new Exception(“Invalid type”);
}
sendMessage($message);
```

Now I'm happier…

```
if ($type == 1) {
    $message = 'hello';
} elseif ($type == 2) {
    $message = 'goodbye';
} else {
    throw new Exception("Invalid type");
}
sendMessage($message);
```

Won’t our code crash more?

Take away

Be explicit:
Use assertions to document assumptions or limitations

Improving error messages

Improving error messages

Invalid type

Improving error messages

Invalid type

Invalid type [$type]

Improving error messages

Invalid type

Invalid type [$type]

Invalid type [$type] for user [$userId]

Benefits of Assertions

Benefits of Assertions

- Document assumptions / limitations
  - type example
  - roman numeral

Benefits of Assertions

- Document assumptions / limitations
  - type example
  - roman numeral
- NOT validation

Benefits of Assertions

- Document assumptions / limitations
  - type example
  - roman numeral
- NOT validation
- Messages to other developers / future you

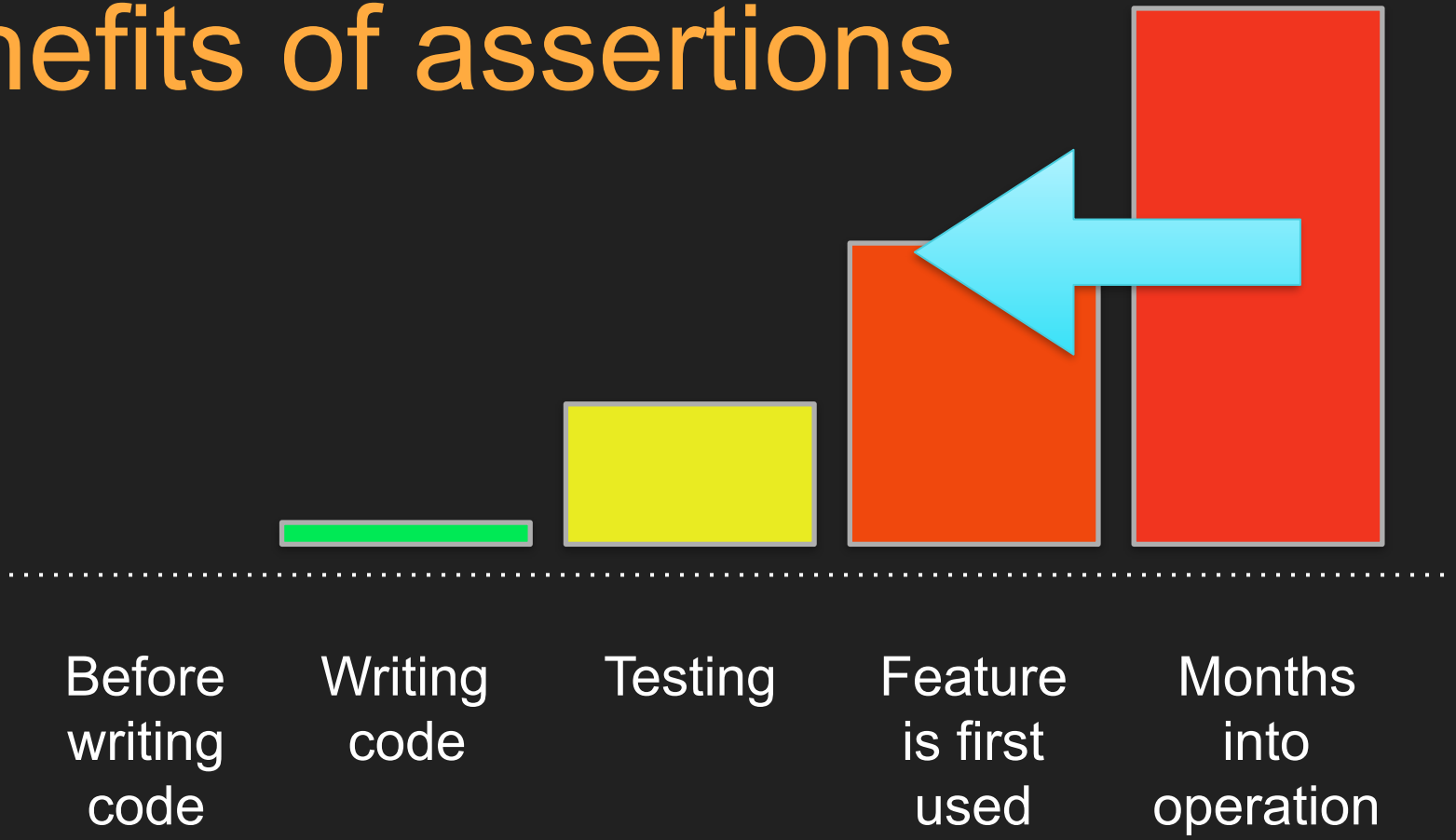
Benefits of Assertions

- Document assumptions / limitations
  - type example
  - roman numeral
- NOT validation
- Messages to other developers / future you
- Can not be ignored

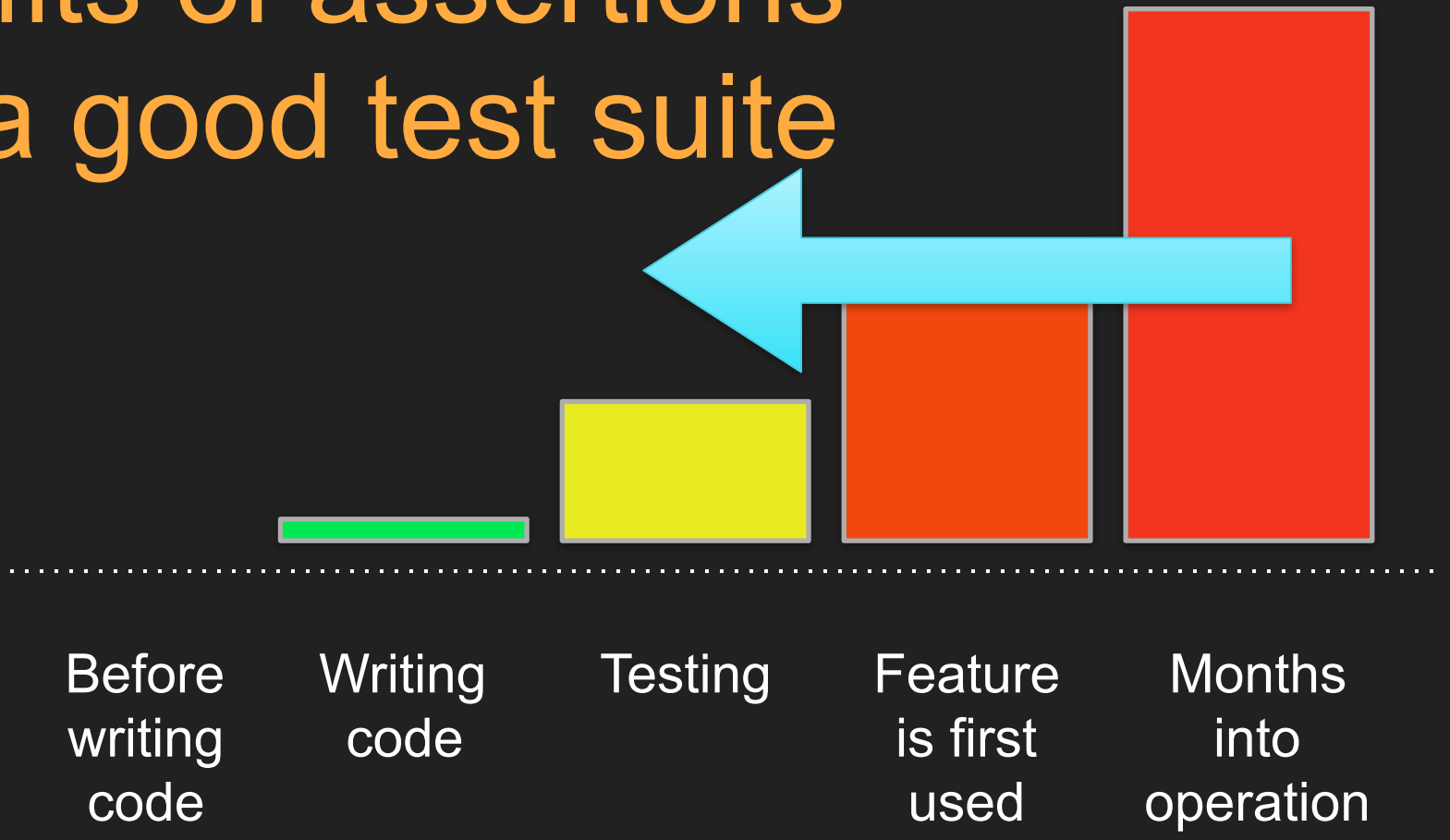
Benefits of Assertions

- Document assumptions / limitations
  - type example
  - roman numeral
- NOT validation
- Messages to other developers / future you
- Can not be ignored
- Sooner you fail the better

Benefits of assertions



Benefits of assertions with a good test suite



Run time analysis recap

Run time analysis recap

- Code is executed

Run time analysis recap

- Code is executed
- Only find bugs in code that is executed

Run time analysis recap

- Code is executed
- Only find bugs in code that is executed
- Assertions document assumptions and limitations

Obvious code

Obvious code

- Use objects rather than primitives

Obvious code

- Use objects rather than primitives
- Readable code

Obvious code

- Use objects rather than primitives
- Readable code
- Rename and refactor

Don’t be so primitive

Can we improve this code?

```
class MarketingCampaign {

  … some methods …

  public function addAddress(string $address);
}


$campaign = new MarketingCampaign();
$campaign->addAddress(“dave@phpsw.uk”)
```

Can we improve this code?

```
class MarketingCampaign {

   … some methods …

   public function addAddress(string $address);
}


$campaign = new MarketingCampaign();
$campaign->addAddress(“dave@phpsw.uk”)
```

These are all strings…

dave@phpsw.uk

fredblogs.com

fred.blogs

fred@blogs.com

6 Lower Park Row, Bristol

These are all strings…

dave@phpsw.uk

fredblogs.com

fred.blogs

fred@blogs.com

6 Lower Park Row, Bristol

This is wrong (and our IDE can’t spot mistake)

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(string $address);
}


$campaign = new MarketingCampaign();
$campaign->addAddress(“6 Lower Park Row, Bristol”)
```

This is wrong (and our IDE can’t spot mistake)

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(string $address);
}


$campaign = new MarketingCampaign();
$campaign->addAddress(“6 Lower Park Row, Bristol”)
```

EmailAddress object instead of primitive

```
class EmailAddress {

  private $emailAddress;

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

  public function getEmailAddress(): string {
    return $this->emailAddress;
  }
}
```

EmailAddress object instead of primitive

```
class EmailAddress {

  private $emailAddress;

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

  public function getEmailAddress(): string {
    return $this->emailAddress;
  }
}
```

EmailAddress object instead of primitive

```
class EmailAddress {

  private $emailAddress;

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

  public function getEmailAddress(): string {
    return $this->emailAddress;
  }
}
```

EmailAddress object instead of primitive

```
class EmailAddress {

  private $emailAddress;

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

  public function getEmailAddress(): string {
    return $this->emailAddress;
  }
}
```

Using EmailAddress

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(EmailAddress $address);
}


$campaign = new MarketingCampaign();
$emailAddress = new EmailAddress(“dave@phpsw.uk”)
$campaign->addAddress($emailAddress)
```

Using EmailAddress

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(EmailAddress $address);
}


$campaign = new MarketingCampaign();
$emailAddress = new EmailAddress(“dave@phpsw.uk”)
$campaign->addAddress($emailAddress)
```

Using EmailAddress

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(EmailAddress $address);
}


$campaign = new MarketingCampaign();
$emailAddress = new EmailAddress(“dave@phpsw.uk”)
$campaign->addAddress($emailAddress)
```

Using EmailAddress

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(EmailAddress $address);
}


$campaign = new MarketingCampaign();
$emailAddress = new EmailAddress(“dave@phpsw.uk”)
$campaign->addAddress($emailAddress)
```

This will fail (and your IDE will warn you)

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(EmailAddress $address);
}


$campaign = new MarketingCampaign();
$campaign->addAddress("6 Lower Park Row, Bristol")
```

This will fail (and your IDE will warn you)

```
class MarketingCampaign {

   .. some methods ..

   public function addAddress(EmailAddress $address);
}


$campaign = new MarketingCampaign();
$campaign->addAddress("6 Lower Park Row, Bristol")
```

But this is wrong

```
$emailAddress = new EmailAddress(“6 Lower Park Row”);
```

But this is wrong

```
$emailAddress = new EmailAddress(“6 Lower Park Row”);
```

Add validation

```
public function __construct(string $emailAddress) {

  $isValidEmailAddress = … check valid email …

  Assert::true($isValidEmailAddress,
    “Invalid email address [$emailAddress]”);


  $this->emailAddress = $emailAddress;
}
```

Add validation

```
public function __construct(string $emailAddress) {
    $isValidEmailAddress = … check valid email …

    Assert::true($isValidEmailAddress,
      “Invalid email address [$emailAddress]”);

    $this->emailAddress = $emailAddress;
}
```

Big win

We’re guaranteed that EmailAddress represents a correctly formatted email address.

Take away

Be explicit:

Use objects rather than primitives if it makes sense to do so

More advantages of objects rather than primitives

Are these email addresses the same?

dave@phpsw.uk

DAVE@phpsw.uk

DAVE@phpsw.UK

dave@PHPSW.uk

Store canonical form

```
public function __construct(string $emailAddress) {

  … validate email address …

  $this->emailAddress = $this->asCanonical($emailAddress);
}
```

Store canonical form

```
public function __construct(string $emailAddress) {

   … validate email address …

   $this->emailAddress = $this->asCanonical($emailAddress);
}
```

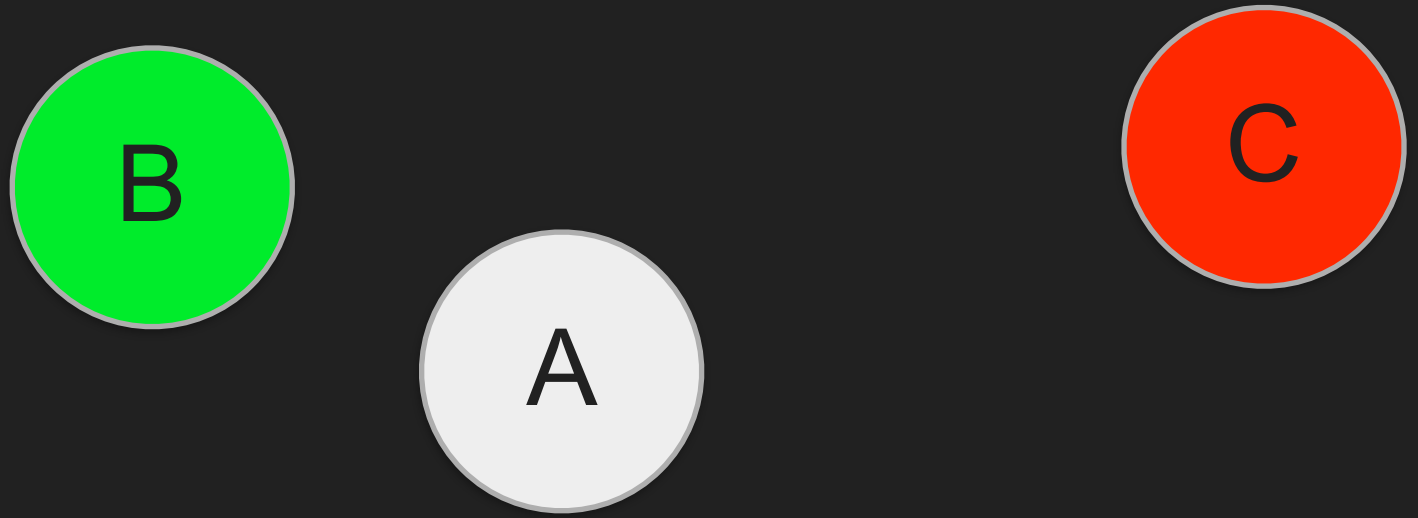
Postcodes formats

Canonical:	`B1 1AB`
No spaces:	`B11AB`
Fixed width:	`B1  1AB`

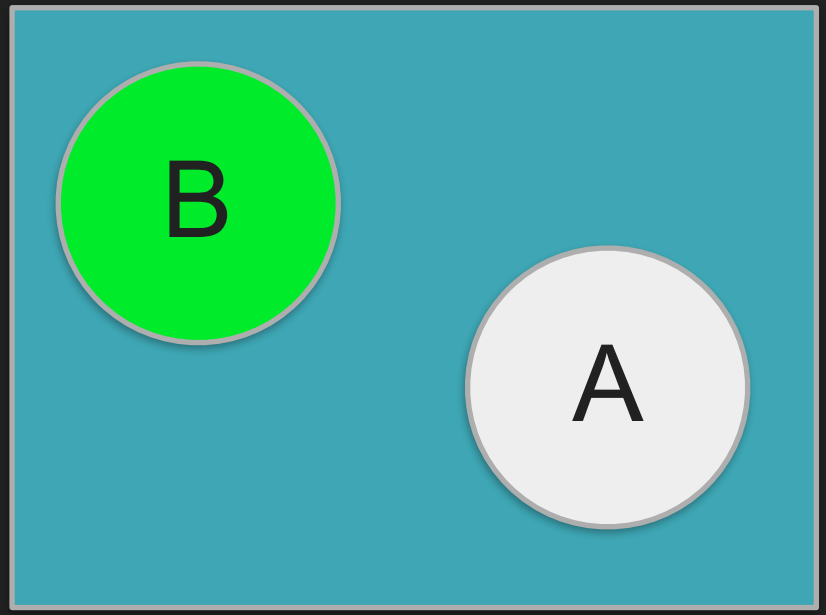
Add domain specific logic

```
public function getPostcode(): string  {…}

public function getNoSpacesPostcode(): string {…}

public function getFixedWidthPostcode(): string {…}
```

Are these positions equal?



Are these positions equal?



Add equals method

```
class Point
{
   const TOLERANCE = 10;

   … Other methods …

   public function equals(Point $other): bool
   {
      $distance = calculateDistance($this, $other);
      return $distance < self::TOLERANCE;
   }
}
```

Add equals method

```
class Point
{
    const TOLERANCE = 10;

    … Other methods …

    public function equals(Point $other): bool
    {
        $distance = calculateDistance($this, $other);
        return $distance < self::TOLERANCE;
    }
}
```

Be careful comparing objects…

```
if ($point1 == $point2) {
    … some code …
}


if ($point1->equals($point2)) {
    … some code …
}
```

Be careful comparing objects…

```
if ($point1 == $point2) {
    … some code …
}
```

```
if ($point1->equals($point2)) {
    … some code …
}
```

Be careful comparing objects…

```
if ($point1 == $point2) {
    … some code …
}
```

```
if ($point1->equals($point2)) {
    … some code …
}
```

Boundaries

```
class Person
{
  /**
   * @Column(type="string")
   */
  private $emailAddress;

  public function setEmailAddress(EmailAddress $emailAddress) {
    $this->emailAddress = $emailAddress->asString();
  }

  public function getEmailAddress(): EmailAddress {
     return new EmailAddress($emailAddress);
  }
```

Boundaries

```
class Person
{
    /**
     * @Column(type="string")
     */
    private $emailAddress;

    public function setEmailAddress(EmailAddress $emailAddress) {
        $this->emailAddress = $emailAddress->asString();
    }

    public function getEmailAddress(): EmailAddress {
        return new EmailAddress($emailAddress);
    }
```

Boundaries

```
class Person
{
  /**
   * @Column(type="string")
   */
  private $emailAddress;

  public function setEmailAddress(EmailAddress $emailAddress) {
    $this->emailAddress = $emailAddress->asString();
  }

  public function getEmailAddress(): EmailAddress {
     return new EmailAddress($emailAddress);
  }
```

Boundaries

```
class Person
{
  /**
   * @Column(type="string")
   */
  private $emailAddress;

  public function setEmailAddress(EmailAddress $emailAddress) {
    $this->emailAddress = $emailAddress->asString();
  }
```

```
  public function getEmailAddress(): EmailAddress {
     return new EmailAddress($emailAddress);
  }
```

Benefits of objects over primitives

Before writing code

Writing code

Testing

Feature is first used

Months into operation

Benefits of objects over primitives

Benefits of objects over primitives

- More obvious code

Benefits of objects over primitives

- More obvious code
- Type hinting

Benefits of objects over primitives

- More obvious code
- Type hinting
- Validation

Benefits of objects over primitives

- More obvious code
- Type hinting
- Validation
- Define what equals means

Benefits of objects over primitives

- More obvious code
- Type hinting
- Validation
- Define what equals means
- Add domain specific functionality

Readable code

Is this code correct?

```
function isCatA($a)
{
  if ($a == 18 || $a == 19) {
    return true;
  }
  return false;
}
```

All tests pass. 100% line, branch & boolean coverage. Test are sensible.

Is this code correct?

```
function isCategoryAdult($a)
{
   if ($a == 18 || $a == 19) {
      return true;
   }
   return false;
}
```

All tests pass. 100% line, branch & boolean coverage. Test are sensible.

Is this code correct?

```
function isCategoryAdult($age)
{
   if ($age == 18 || $age == 19) {
      return true;
   }
   return false;
}
```

All tests pass. 100% line, branch & boolean coverage. Test are sensible.

Is this code correct?

```
function isCategoryAdult(int $age): bool
{
    if ($age == 18 || $age == 19) {
        return true;
    }
    return false;
}
```

All tests pass. 100% line, branch & boolean coverage. Test are sensible.

Is this code correct?

```
/**
 * Return true if person is adult
 * (age is 18 or over)
 */
function isCategoryAdult(int $age): bool
{
 if ($age == 18 || $age == 19) {
    return true;
  }
  return false;
}
```

Test cases

Age	Expected output
17	false
18	true
19	true

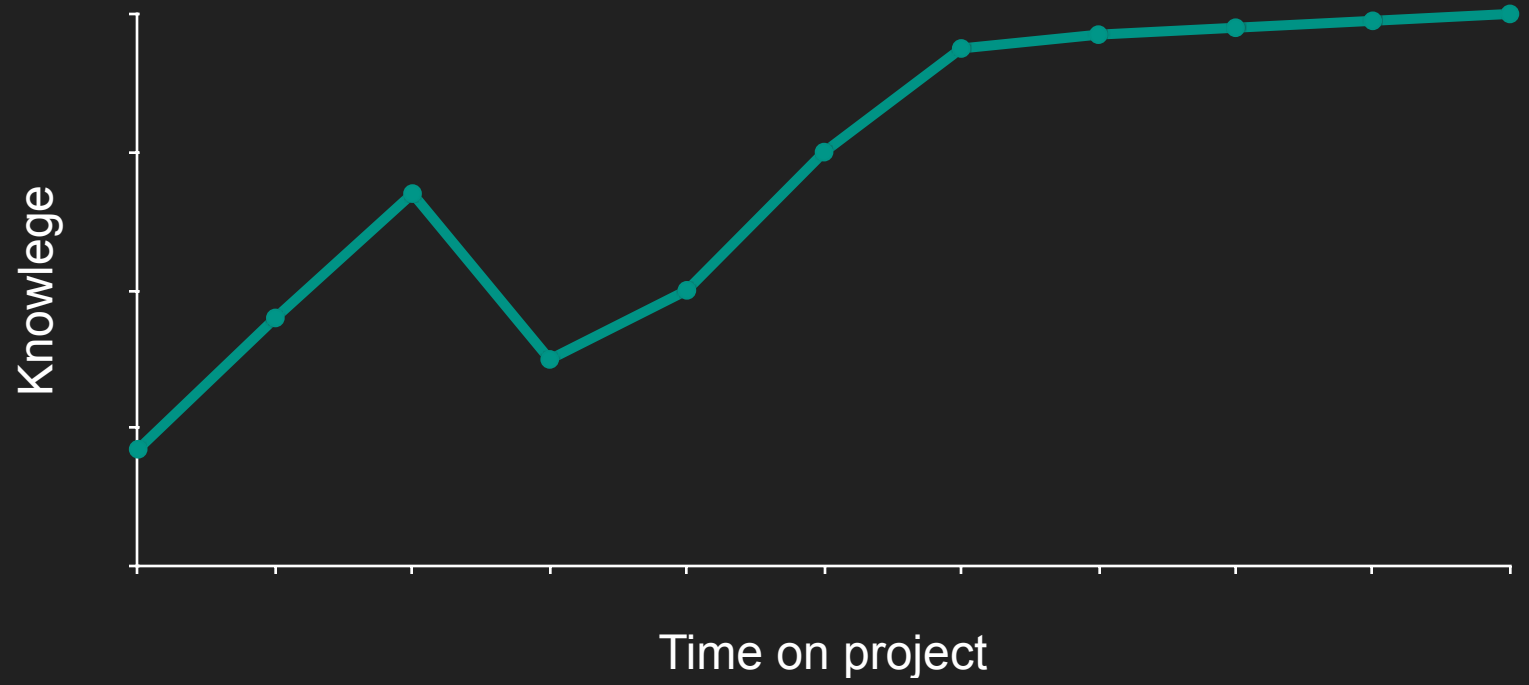
Could we write more tests?

It’s harder for bugs to hide in clean code

Take away

Be explicit:
Write really obvious, really boring code.

Renaming


Knowlege
Time on project

Renaming

```
class User {
 public function getName() {…}
}


class Game {
 public function getName() {…}
}
```

Renaming

```
class User {
 public function getName() {…}
}


class Game {
 public function getName() {…}
}
```

Renaming

```
class User {
 public function getName() {…}
}


class Game {
 public function getQuest() {…}
}
```

Renaming

```
class User {
 public function getName() {…}
}


class Game {
 public function getQuest() {…}
}
```

Renaming

```
function getUser();

function getGame();


$user = getUser();
$game = getGame();

echo ‘Hello ‘ . $user->getName();
echo ‘You are playing ‘ . $game->getName();
```

Renaming

```
function getUser();

function getGame();


$user = getUser();
$game = getGame();

echo ‘Hello ‘ . $user->getName();
echo ‘You are playing ‘ . $game->getName();
```

Renaming

```
function getUser(): User;

function getGame(): Game;


$user = getUser();
$game = getGame();

echo ‘Hello ‘ . $user->getName();
echo ‘You are playing ‘ . $game->getQuest();
```

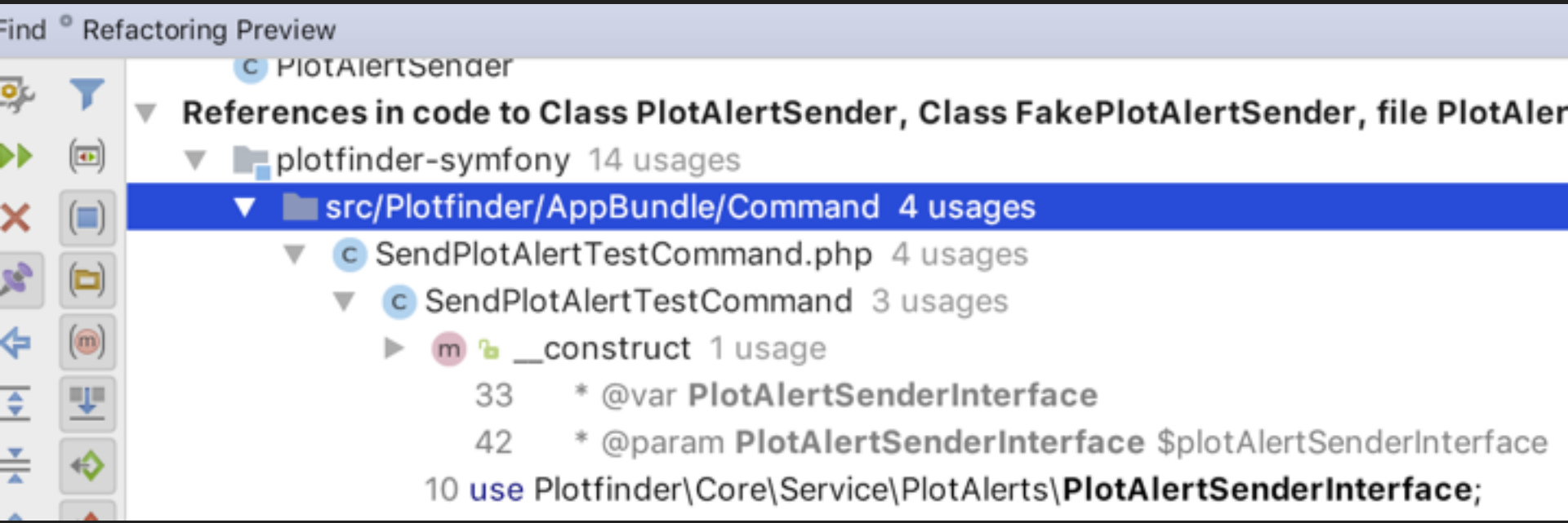
Renaming

```
function getUser(): User;

function getGame(): Game;


$user = getUser();
$game = getGame();

echo ‘Hello ‘ . $user->getName();
echo ‘You are playing ‘ . $game->getQuest();
```

Renaming

```
function getUser(): User;

function getGame(): Game;


$user = getUser();
$game = getGame();

echo 'Hello ' . $user->getName();
echo 'You are playing ' . $game->getQuest();
```

Win-Win: Rename and refactor



Take away

Use your IDE to rename code to be cleaner

Refactor

Assertions are good, but some could point to a code smell

Possible code smell

```
$shape = new Shape();
$shape->setType(“square”)
```

Possible code smell

```
switch($shape->getType()) {
  case ‘square’:
    $area = … calculate area of square …
    break;
  case ‘triangle’:
    $area = … calculate area of triangle …
    break;
  … more shapes …
  default:
    throw new Exception(“Invalid shape”);
}
```

Possible code smell

```
switch($shape->getType()) {
  case ‘square’:
    $area = … calculate area of square …
    break;
  case ‘triangle’:
    $area = … calculate area of triangle …
    break;
  … more shapes …
  default:
    throw new Exception(“Invalid shape”);
}
```

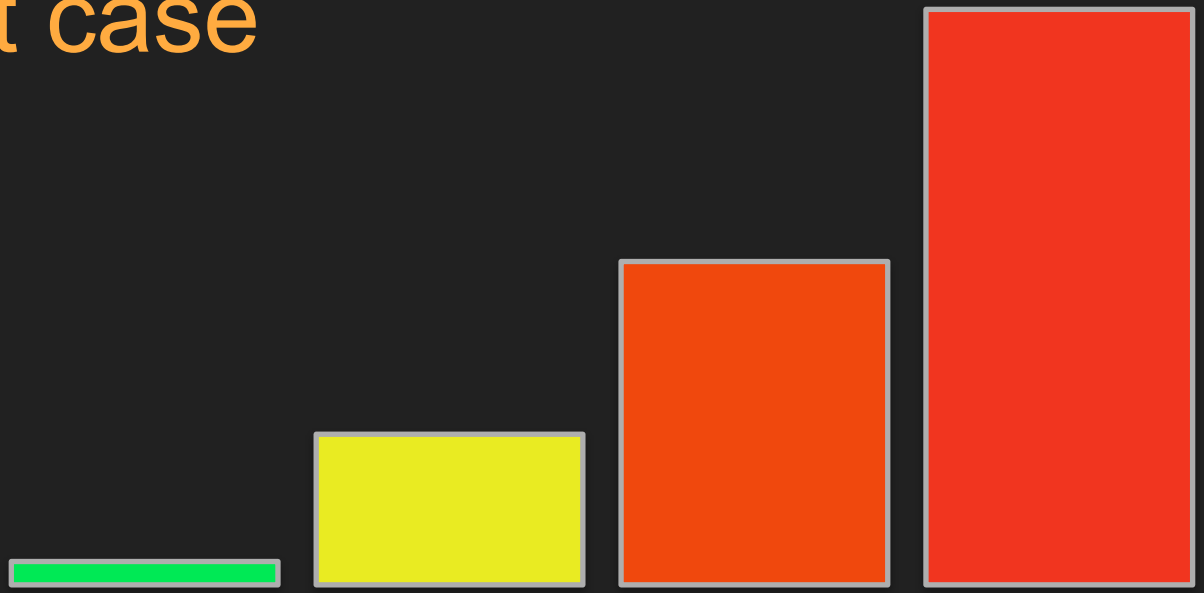
Possible code smell

```
switch($shape->getType()) {
  case ‘square’:
    $area = … calculate area of square …
    break;
  case ‘triangle’:
    $area = … calculate area of triangle …
    break;

  … more shapes …

  default:
    throw new Exception(“Invalid shape”);
}
```

Possible code smell

```
switch($shape->getType()) {
  case ‘square’:
    $area = … calculate area of square …
    break;
  case ‘triangle’:
    $area = … calculate area of triangle …
    break;
  … more shapes …
  default:
    throw new Exception(“Invalid shape”);
}
```

Best case



Before writing code

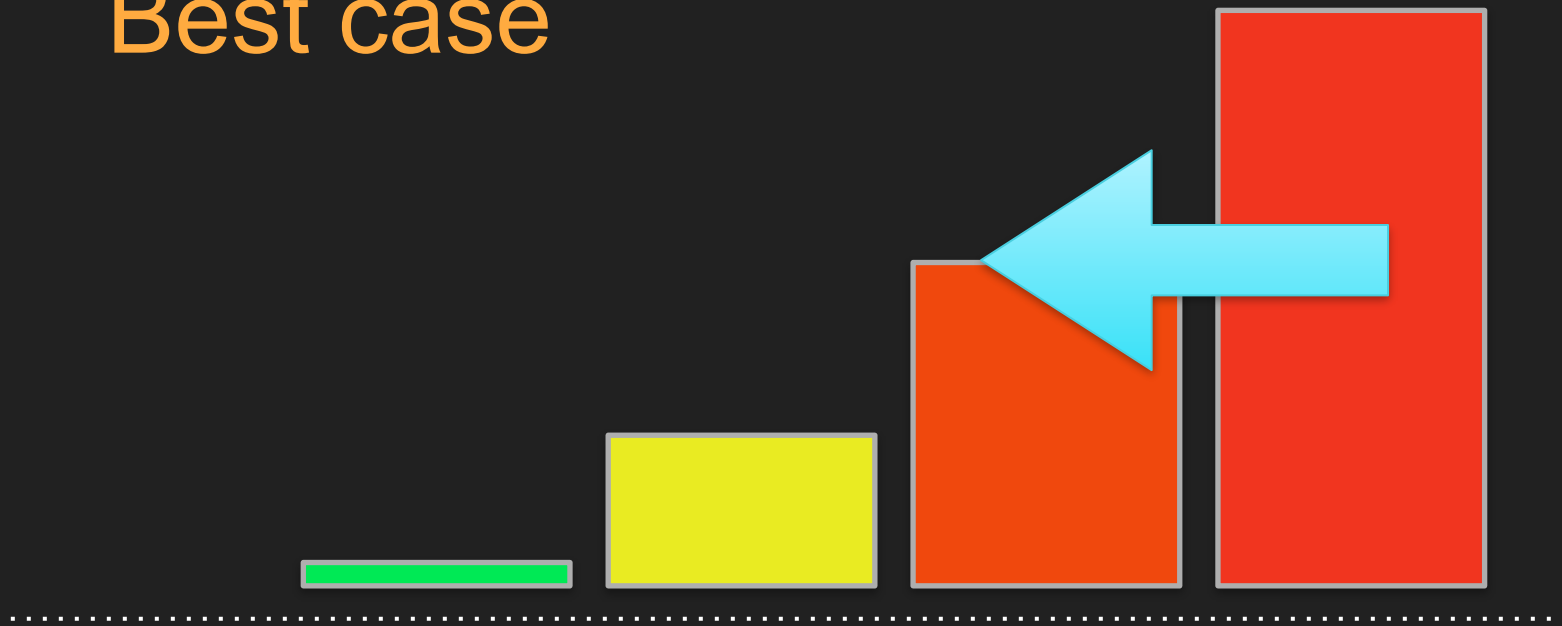
Writing code

Testing

Feature is first used

Months into operation

Best case



Before writing code

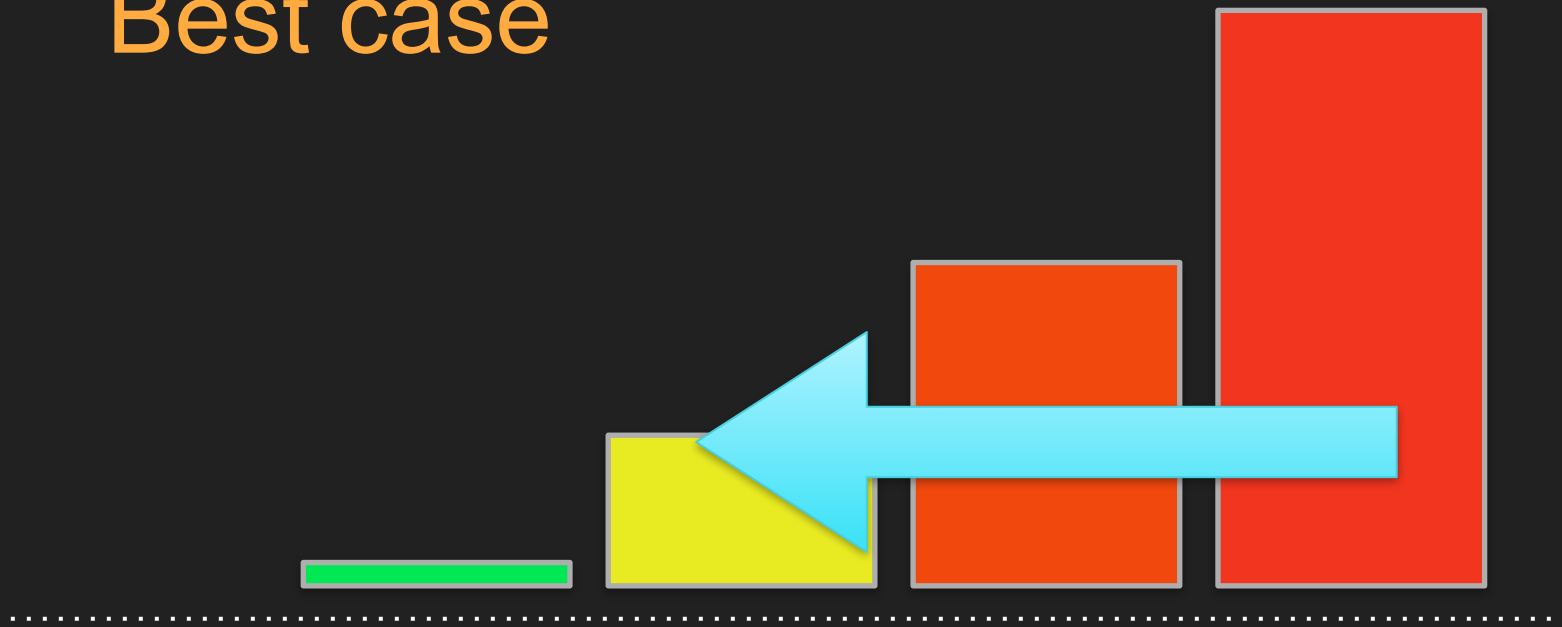
Writing code

Testing

Feature is first used

Months into operation

Best case



Before writing code

Writing code

Testing

Feature is first used

Months into operation

Improve the shape object

```
class Shape {

   public function getType() {…}



}
```

Improve the shape object

```
abstract class Shape {

    public function getType() {…}


}
```

Improve the shape object

```
abstract class Shape {

    public function getType() {…}

    abstract public function getArea();

}
```

Make an class for each shape

```
class Square extends Shape {

   public function getArea() {
      … calculate area of square …
   }

}
```

Make an class for each shape

```
class Square extends Shape {

   public function getArea() {
      … calculate area of square …
   }

}
```

Make an class for each shape

```
class Square extends Shape {

    public function getArea() {
        … calculate area of square …
    }

}
```

Make an class for each shape

```
class Triangle extends Shape {

   public function getArea() {
      … calculate area of triangle …
   }

}
```

We replace…

```
$shape = new Shape(“square”);
```

With…

```
$shape = new Square();
```

And replace…

```
switch($shape->getType()) {
  case ‘square’:
    $area = … calculate area of square …
    break;
  case ‘triangle’:
    $area = … calculate area of triangle …
    break;
  … more shapes …
  default:
    throw new Exception(“Invalid shape”);
}
```

With this…

```
$area = $shape->getArea();
```

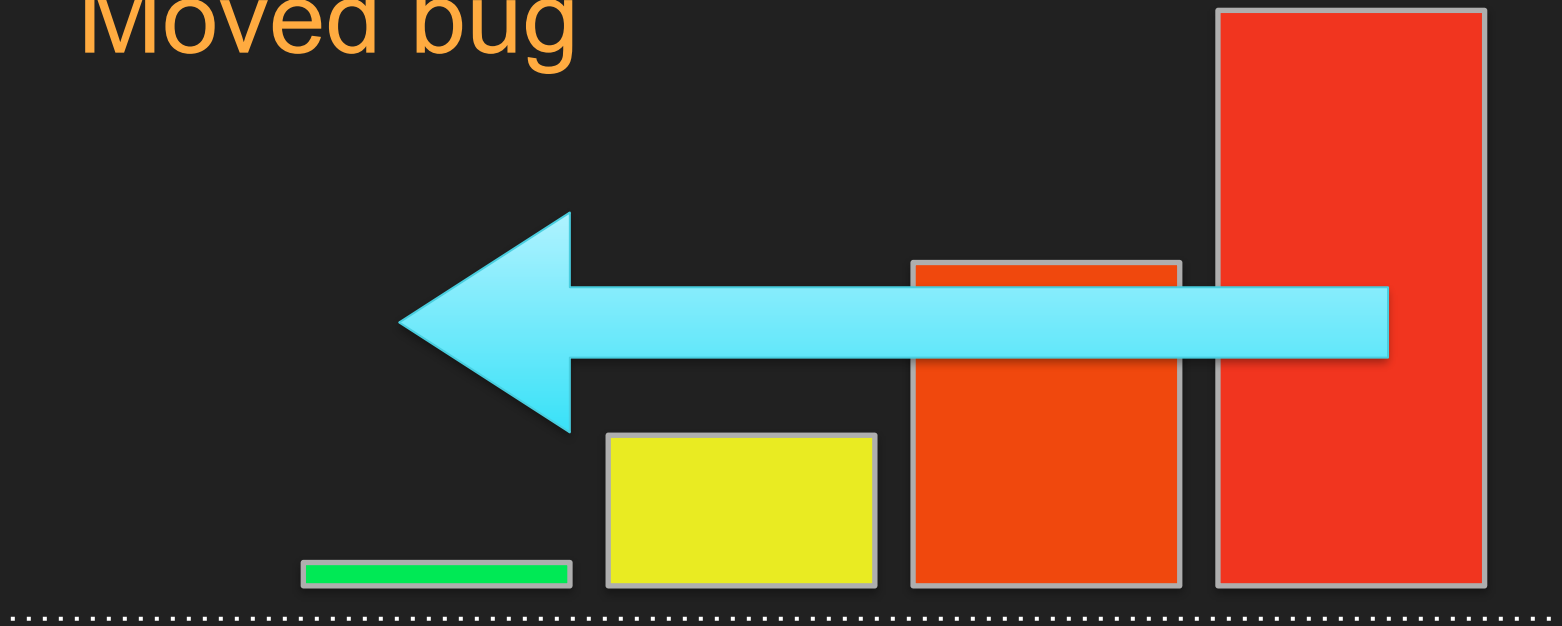
Introduce a new shape

```
class Hexagon extends Shape {

   public function getArea() {
      … calculate area of hexagon …
   }

}
```

Introduce a new shape

```
class Hexagon extends Shape {

    public function getArea() {
        … calculate area of hexagon …
    }

}
```

Moved bug



Before writing code

Writing code

Testing

Feature is first used

Months into operation

Take away

Refactor code to be cleaner

Obvious code recap

Obvious code recap

- Clean code = hard for bugs to hide

Obvious code recap

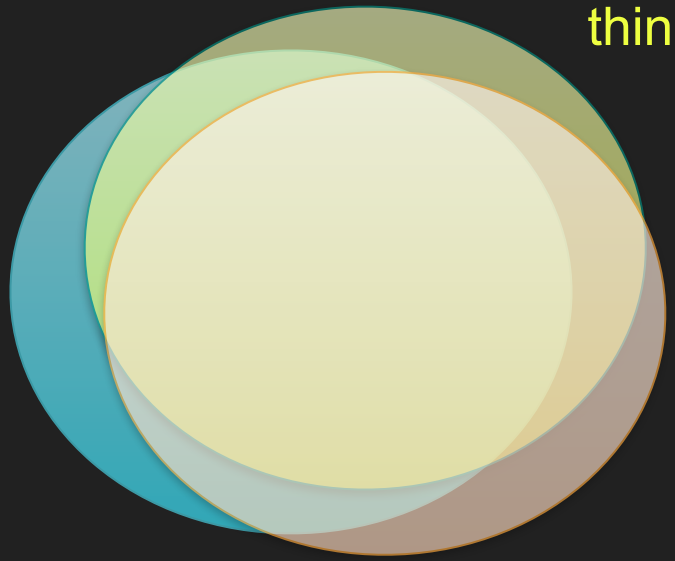
- Clean code = hard for bugs to hide
- Consider using objects rather than primitives

Obvious code recap

- Clean code = hard for bugs to hide
- Consider using objects rather than primitives
- Rename and refactor to keep your code clean

Summary: Be more explicit

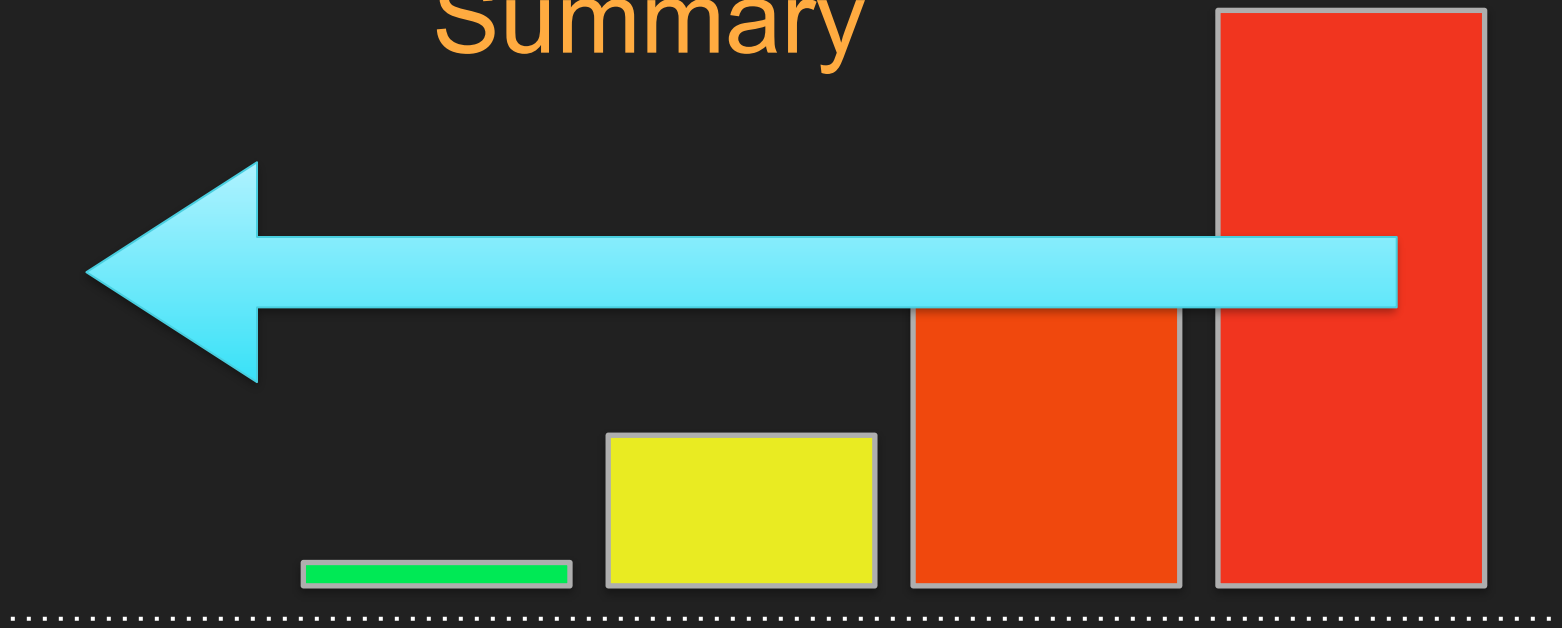
What the developer thinks the code does



What the code should do

What the code actually does

Summary



Before writing code

Writing code

Testing

Feature is first used

Months into operation

Summary

Summary

- Type hint everything you can

Summary

- Type hint everything you can
- Use docblock for language gaps

Summary

- Type hint everything you can
- Use docblock for language gaps
- Write tests

Summary

- Type hint everything you can
- Use docblock for language gaps
- Write tests
- Add assertions

Summary

- Type hint everything you can
- Use docblock for language gaps
- Write tests
- Add assertions
- Use objects over primitives

Summary

- Type hint everything you can
- Use docblock for language gaps
- Write tests
- Add assertions
- Use objects over primitives
- Rename and refactor

Summary

- Use a modern IDE

Questions

Advice for improving

https://joind.in/talk/84cb4