TEST AUTOMATICI CON PHPUNIT

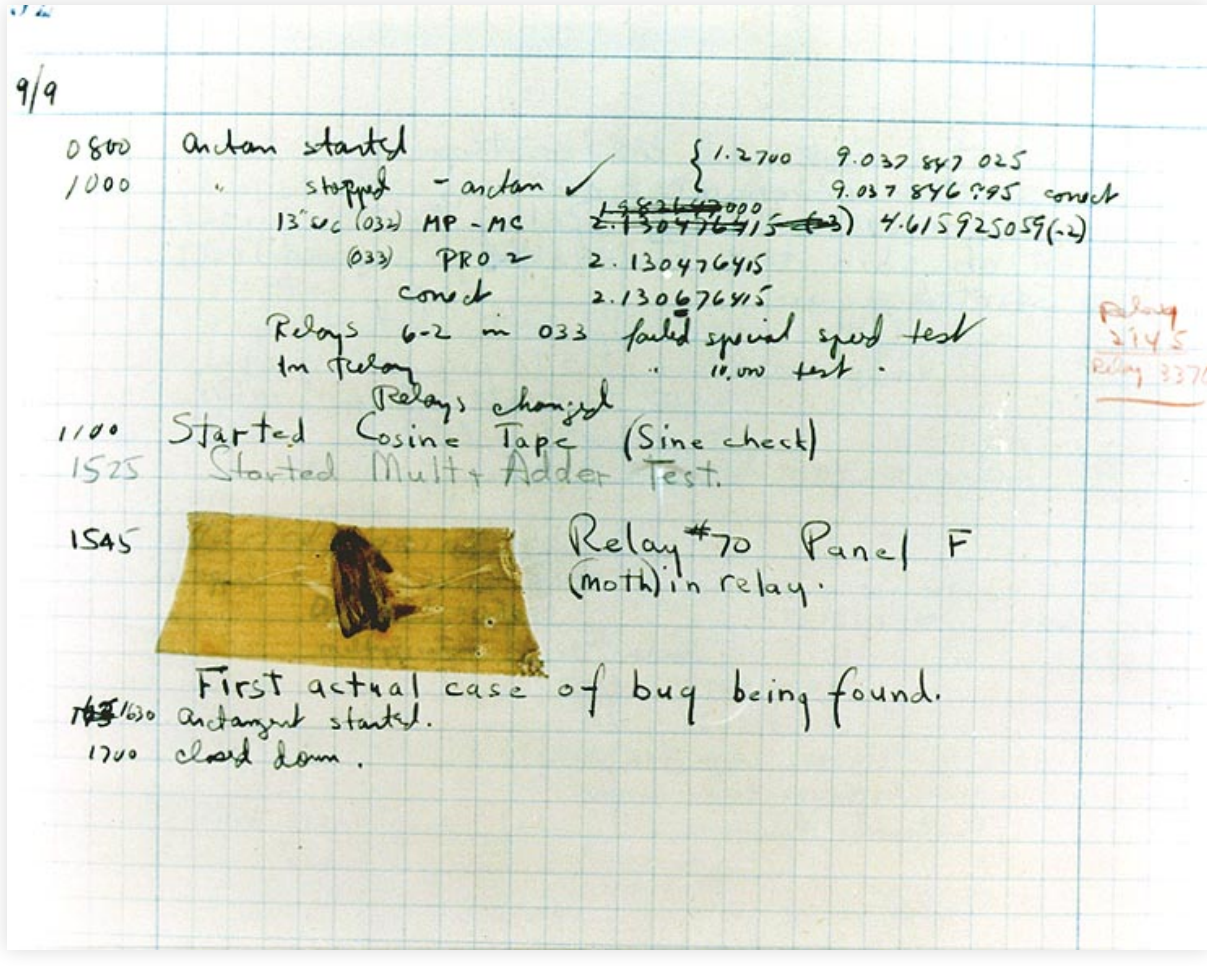
Corso Backend System Integrator

Modulo PHP Programming

Dott. Enrico Zimuel

PROGRAMMA

- Test automatici
- PHPUnit
- Configurazione
- setUp/tearDown
- Data provider
- Mock di oggetti
- Code coverage



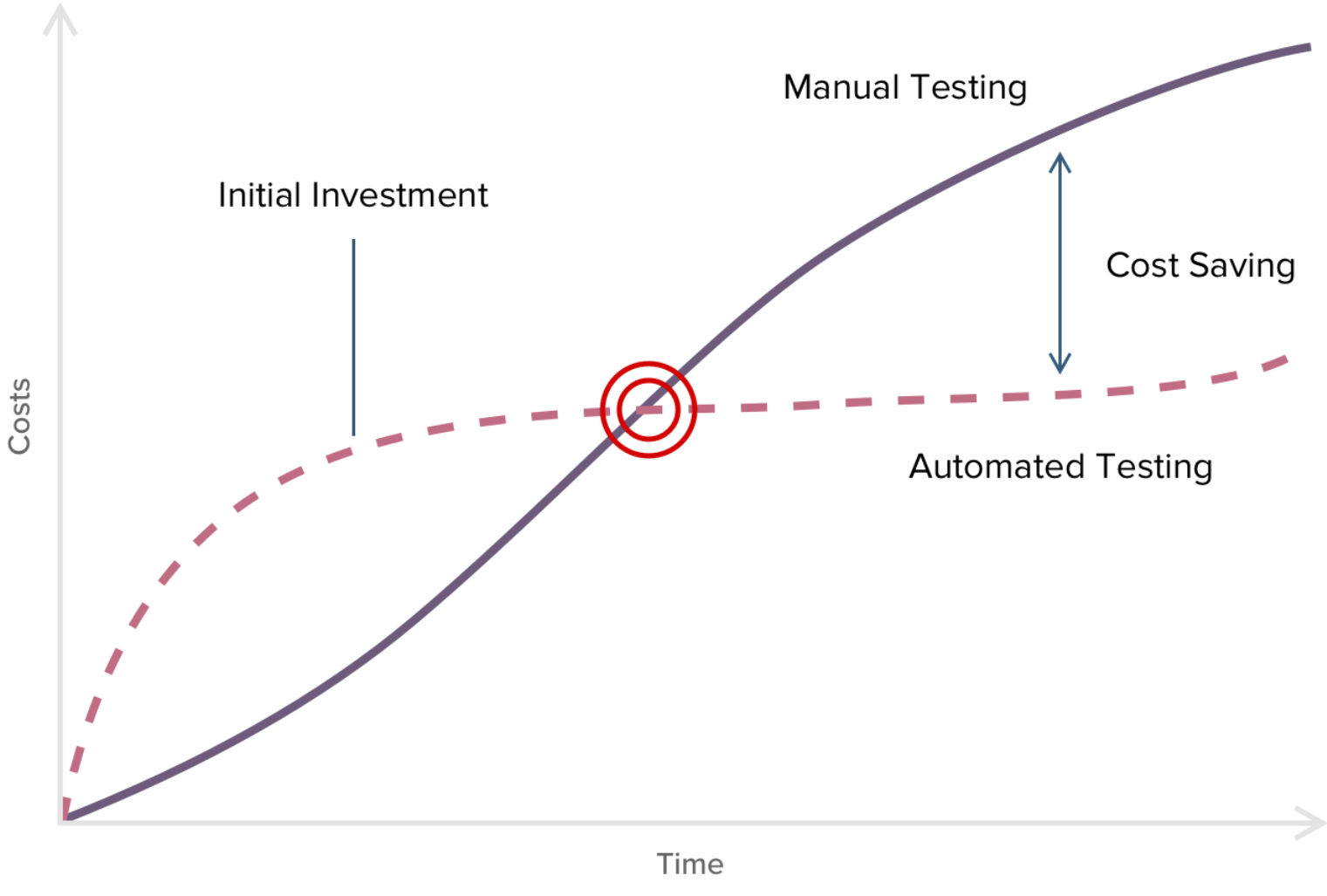
Harvard Mark II electromechanical computer's log

TEST AUTOMATICI

I test automatici sono una collezione di asserzioni, ossia una serie di istruzioni che testano la corretta esecuzione di una porzione di codice, utilizzando dei dati in ingresso prestabiliti e verificando che i risultati siano quelli attesi

IMPORTANZA DEI TEST

- Riducono il numero di bug
- Riducono i tempi di sviluppo (a lungo termine)
- Aiutano a chiarire le specifiche del progetto
- Aumentano la confidenza nel codice, e.s. durante il refactoring
- Garantiscono notti tranquille agli sviluppatori


Manual Testing
Initial Investment
Cost Saving
Automated Testing
Costs
Time

SOFTWARE FAILURE (ESEMPI)

- NASA's Mars Climate Orbiter, $128 milioni andati in fumo per un errore di conversione metrico
- Pentium FDIV bug, un bug del processore Pentium nella divisione in virgola mobile. Danni per $475 milioni
- Ariane 5 Flight 501, un satellite esploso pochi secondi dopo il decollo a causa di un overflow di 64-bit in 16-bit. Danni per $8 bilioni


PHPUnit

PHPUNIT

- Libreria PHP per sviluppare test automatici
- Progetto open source nato nel 2004 per opera di Sebastian Bergman
- Standard de facto per i test automatici in PHP
- Sito ufficiale del progetto: phpunit.de

INSTALLAZIONE

```
composer require --dev phpunit/phpunit
```

Installa phpunit solo per l'ambiente **dev**

```
{
    "require-dev": {
        "phpunit/phpunit": "^8.4"
    }
}
```

CONFIGURAZIONE

PHPUnit può essere configurato tramite un file di configurazione XML (phpunit.xml):

```
<phpunit>
  <testsuites>
    <testsuite name="Test automatici">
          <directory>./test</directory>
        </testsuite>
  </testsuites>
  <filter>
    <whitelist processuncoveredfilesfromwhitelist="true">
      <directory suffix=".php">./src</directory>
    </whitelist>
  </filter>
</phpunit>
```

STRUTTURA DI UN TEST

- I test in PHPUnit sono raggruppati in classi, denominate con il suffisso **Test** (es. FilterTest)
- Ogni classe deve estendere **PHPUnit\Framework\TestCase**
- Un test è una funzione della classe, denominato con il prefisso **test** (es. testEmailIsValid)

UTILIZZO

PHPUnit è eseguito dal terminale, utilizzando il seguente comando:

```
vendor/bin/phpunit
```

ESEMPIO DI CLASSE DA TESTARE

```php
namespace App;

class Filter
{
    public function isEmail(string $email)
    {
        // @todo da implementare
    }
}
```

ESEMPIO DI TEST

```php
namespace App\Test;
use PHPUnit\Framework\TestCase;
use App\Filter;

class FilterTest extends TestCase
{
    public function testValidEmail() {
        $filter = new Filter();
        $this->assertTrue($filter->isEmail('foo@bar.com'));
    }
    public function testInvalidEmail() {
        $filter = new Filter();
        $this->assertFalse($filter->isEmail('foo'));
    }
}
```

ASSERZIONI

PHPUnit offre numerose asserzioni:

- **assertTrue()**: verifica che l'elemento sia true
- **assertFalse()**: verifica che l'elemento sia false
- **assertEmpty()**: verifica che l'elemento sia vuoto
- **assertEquals()**: verifica che due elementi siano uguali

ASSERZIONI (2)

- **assertGreaterThan()**: verifica che un elemento sia maggiore di
- **assertContains()**: verifica che una stringa sia contenuta in un'altra o che un elemento sia contenuto in un array
- **assertInstanceOf()**: verifica che un elemento sia istanza di una classe specifica
- Qui la lista completa

ESERCIZIO

Implementare il metodo **App\Filter::isEmail()** e verificare che i test automatici siano tutti positivi

Nota: I sorgenti dell'esercizio si trovano su its.mrooms.net

SETUP

PHPUnit offre un metodo per inizializzare un test:

```
class FilterTest extends TestCase
{
  public function setUp(): void
  {
    $this->filter = new Filter();
  }
  public function testValidEmail()
  {
    $this->assertTrue($this->filter->isEmail('foo@bar.com'));
  }
}
```

TEARDOWN

C'è anche un metodo per resettare un test:

```
class FilterTest extends TestCase
{
    public function setUp(): void
    {
        file_put_contents('/tmp/foo', 'Test');
    }
    public function tearDown(): void
    {
        unlink('/tmp/foo');
    }
    public function testFoo()
    {
        // utilizzo il file temporaneo
    }
}
```

CICLO DI UN TEST

setUp() e tearDown() vengono richiamati prima e dopo ogni test (funzione)

```
class FooTest extends TestCase
{
    public function setUp() {}
    public function tearDown() {}

    public function testUno() {}
    public function testDue() {}
}
// setUp(), testUno(), tearDown()
// setUp(), testDue(), tearDown()
```

DATA PROVIDER

```
class DataTest extends TestCase
{
    /**
     * @dataProvider getDataInput
     */
    public function testAdd($a, $b, $expected) {
        $this->assertEquals($expected, $a + $b);
    }
    public function getDataInput() {
        return [
            [0, 0, 0],
            [0, 1, 1]
        ];
    }
}
```

ESERCIZIO

Modificare la classe FilterTest utilizzando un *data provider* per verificare che i dati **"foo@bar.com"** e **"foo"** siano validi o meno

MOCK DI OGGETTI

- Il mocking di un oggetto è una tecnica che consente di creare una copia fasulla di un oggetto per finalità di testing
- Viene utilizzato per gestire le dipendenze esplicite delle classi
- In un test si evitano di utilizzare dipendenze con istanze reali di oggetti

ESEMPIO

```php
class Foo {
    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }
    public function getAll(): array
    {
        $sth = $this->pdo->prepare('SELECT * FROM table');
        // ...
    }
}
```

ESEMPIO (2)

```
class FooTest extends TestCase
{
    public function setUp()
    {
        $this->pdo = $this->createMock(PDO::class);
        $this->sth = $this->createMock(PDOStatement::class);
        $this->pdo->method('prepare')
            ->willReturn($this->sth);
        $this->foo = new Foo($this->pdo);
    }
    public function testGetAllReturnArray()
    {
        $this->assertIsArray($this->foo->getAll());
    }
}
```

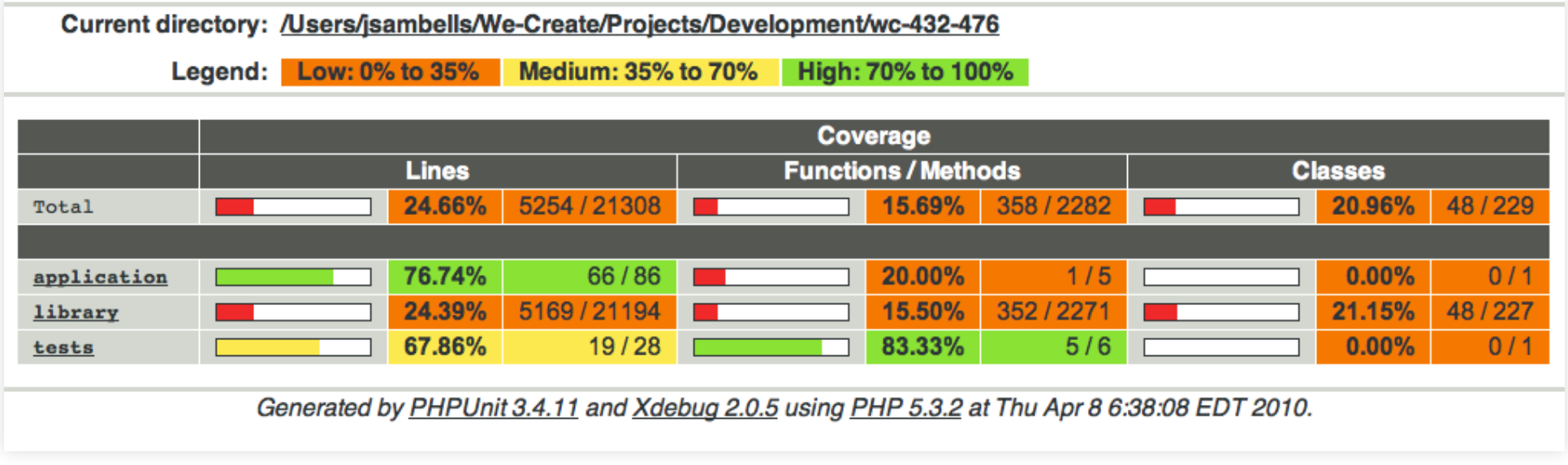
CODE COVERAGE

- La copertura del codice (**code coverage**) è una misura che indica la percentuale, espressa in righe di codice, delle istruzioni eseguite durante un test
- Esempio una copertura dell'80% indica che i test automatici non eseguono il 20% del codice
- PHPUnit offre la possibilità di calcolare la code coverage tramite il comando:

```
vendor/bin/phpunit --coverage-text
```

XDEBUG

Per eseguire la funzionalità di code coverage è necessario installare Xdebug



Current directory: /Users/jsambells/We-Create/Projects/Development/wc-432-476

Legend: Low: 0% to 35% | Medium: 35% to 70% | High: 70% to 100%

	Coverage								
	Lines			Functions / Methods			Classes		
Total		24.66%	5254 / 21308		15.69%	358 / 2282		20.96%	48 / 229
application		76.74%	66 / 86		20.00%	1 / 5		0.00%	0 / 1
library		24.39%	5169 / 21194		15.50%	352 / 2271		21.15%	48 / 227
tests		67.86%	19 / 28		83.33%	5 / 6		0.00%	0 / 1

Generated by PHPUnit 3.4.11 and Xdebug 2.0.5 using PHP 5.3.2 at Thu Apr 8 6:38:08 EDT 2010.

ESERCIZIO

Installare o verificare la presenza di Xdebug ed eseguire il code coverage dell'esercizio precedente

ESERCIZIO (DA CONSEGNARE)

- Scrivere una classe per gestire un sistema di autenticazione basato su email e password
- Questa classe deve implementare la seguente interfaccia:

```php
interface Authenticate
{
    public function verify(string $email, string $password) : bool
}
```

- Le email e le password devono essere memorizzati in un database
- Le password devono essere memorizzate utilizzando la funzione password_hash() del PHP
- Completare l'esercizio scrivendo i test automatici con l'ausilio del PHPUnit

INFORMAZIONI E CONTATTI:

enrico.zimuel@its-ictpiemonte.it