Testing

Jessica Hoeck, Olle Haerstedt

Overview

- Why tests?
- What’s a good test?
- What’s testable code?
- Test paradigms
- Our testing system

Why tests

Reduce risk

Different kind of failures:

- Crash failure (500, 404, white-page)
- Block failure (clicking on button does nothing)
- Invisible failure (clicked “Save” but was in fact not saved)
- Cosmetic failure (spelling mistake, logo not loaded)

Good failure

Visible failure always better than invisible:

- Gets reported at once
- Gets fixed quickly
- Invisible failure often fixed “too late” - already lost or leaked data

Risks

- Bugs are risk
- Unmaintainable or buggy tests are also risk
- Strike balance between too many tests and no tests

Regression

Regression = something worked before; now broken

Regression test = fix bug *and* add test for bug

Goal: to create a test suite and not fix same bug twice

Very important for critical bugs

Refactorization

Test suite makes it safe to refactor the code

Correct way: write the test suite *before* refactoring

Example: EM

Testing paradigms

- Test Doubles
- White Box Testing
- Black Box Testing
- Grey Box Testing
- Test Driven Development (TDD)
- Behavior Driven Development (BDD)
- Risk-based testing

Testing paradigms: unit testing general info

- System Under Test (SUT)
  - Class/Object/Function which will be tested
- Dependencies/ Collaborators
  - SUT iteracts with other part of the system
- Test Doubles
  - simulate dependencies/collaborators with behavior
  - allows unit testing in isolation
- Example Framework:
  - sinon.js (js) for spies,stubs and mocks.

Test Doubles: Mocks

Test Doubles: Dummies

- passed around but never actually used
- common use case:
  - fill parameter list
- Object implements interface nothing else

Test Doubles: Dummies Example

```
var TaskManager = function(){
    var taskList = [];

    return {
        addTask: function(task){
            taskList.push(task);
        },
        tasksCount: function(){
            return taskList.length;
        }
    }
}

// Test
var assert = require("assert")
describe('add task', function(){
    it('should keep track of the number of tasks', function(){
      var DummyTask = function(){ return {} };
      var taskManager = new TaskManager();

      taskManager.addTask(new DummyTask());
      taskManager.addTask(new DummyTask());

      assert.equal( taskManager.tasksCount(), 2 );

    })
})
```

Test Doubles: Spies

- Object that records interaction with other objects
- useful for testing callbacks or how methods are used through SUT

```
"test should call subscribers on publish": function () {
    var callback = sinon.spy();
    PubSub.subscribe("message", callback)
    PubSub.publishSync("message");

    assertTrue(callback.called);
}
```

Test Doubles: Stubs

- fake objects with pre-programmed behavior (simulation)
  - for example: returning fixed values
- typical reasons
  - avoid inconvenient interface - for example: avoid making requests to server from tests
  - feed the system with data

```
"example of simple stub without any lib": function () {
    var task = { completed = true }
}
```

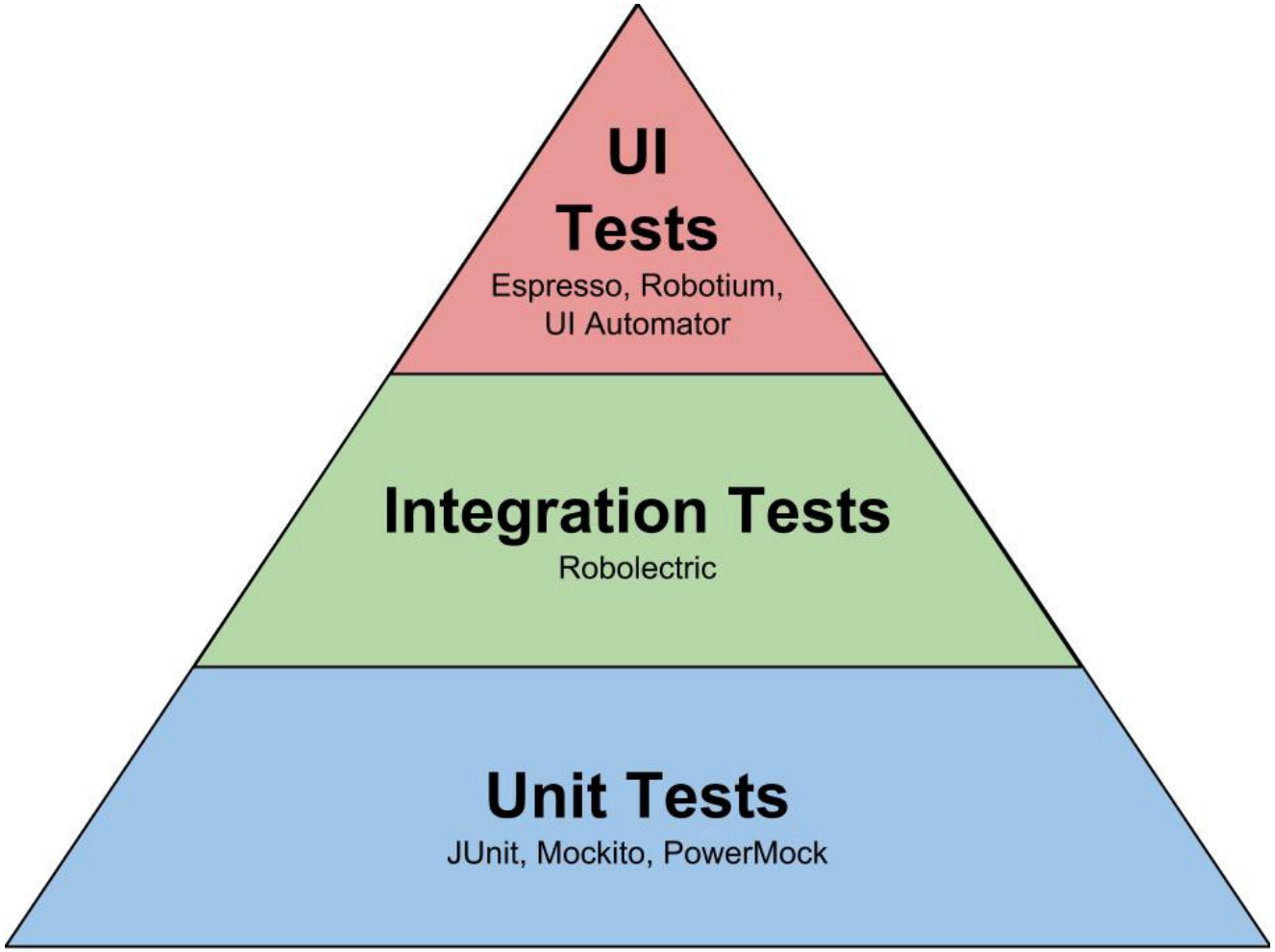
Test Doubles: Fakes

- objects have working implementation, but take shortcuts to make them not suitable for production
  - example: memory database
- One step up from a Stub.
  - it returns values but also works as real collaborator

Test Doubles: Fakes Example

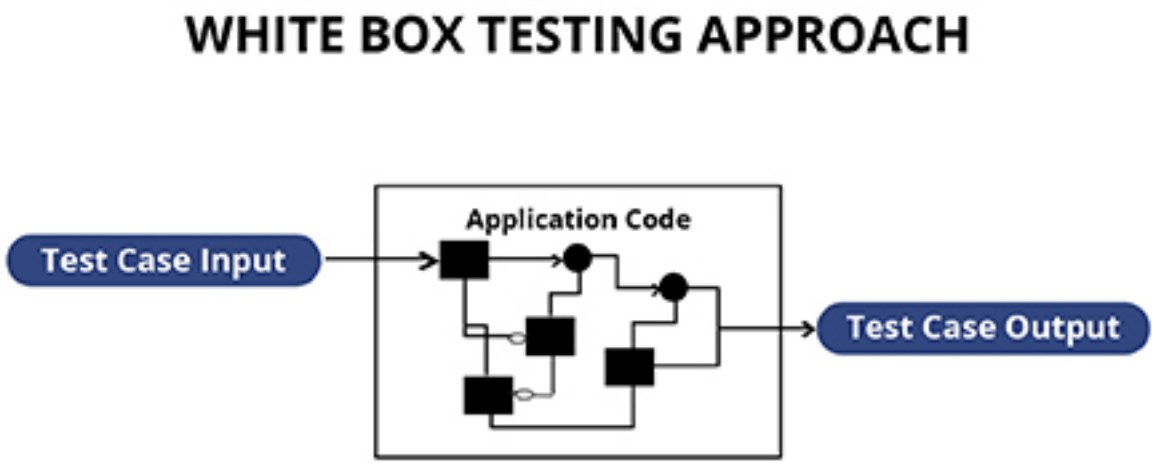
```
var xhr, requests;

before(function  {
    xhr = sinon.useFakeXMLHttpRequest();
    requests = [];
    xhr.onCreate = function (req) { requests.push(req); };
});

after(function () {
    // we must clean up when tampering with globals.
    xhr.restore();
});

it("makes a GET request for todo items", function () {
    getTodos(42, sinon.spy());

    assert.equals(requests.length, 1);
    assert.match(requests[0].url, "/todo/42/items");
});
```

Testing pyramid



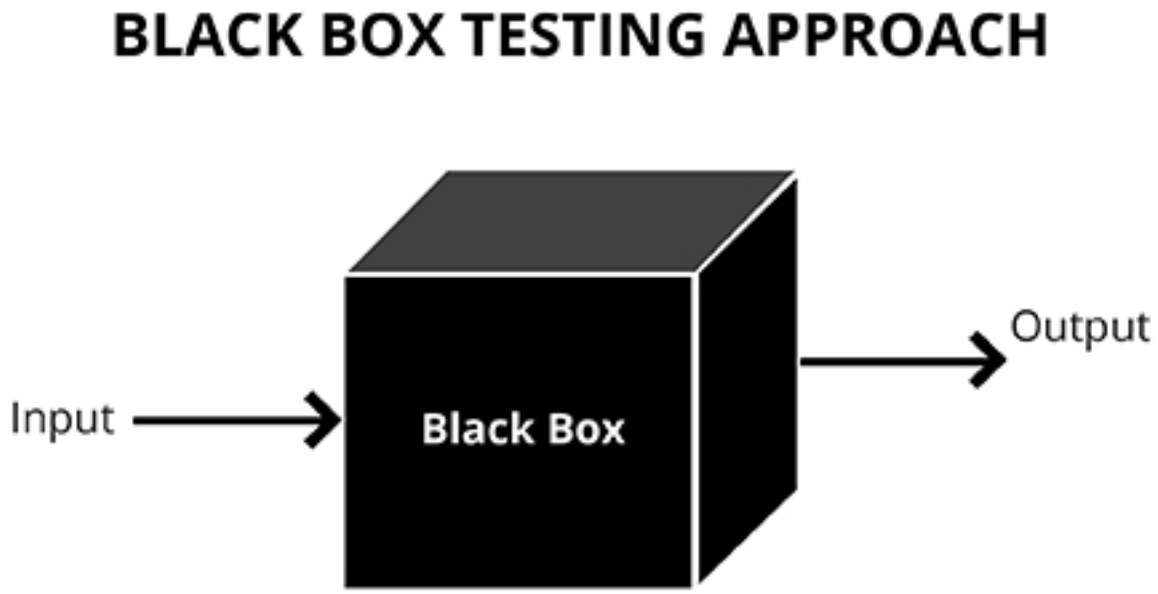
White-box Testing

- Tests internal structures of an application
- Can be applied
  - unit tests
  - integration tests
  - system tests



Black-box Testing

- Method of software testing
- Examines functionality without going into internal structure
- Applied to every kind of tests
  - unit tests ?
  - integration tests
  - system tests
  - acceptance tests



Grey-box Testing

- Combination of white-box testing + black-box testing
- Aim of testing: search defects
  - improper structure
  - improper usage of application

White box vs Black box

- spagetti code = black box testing
-

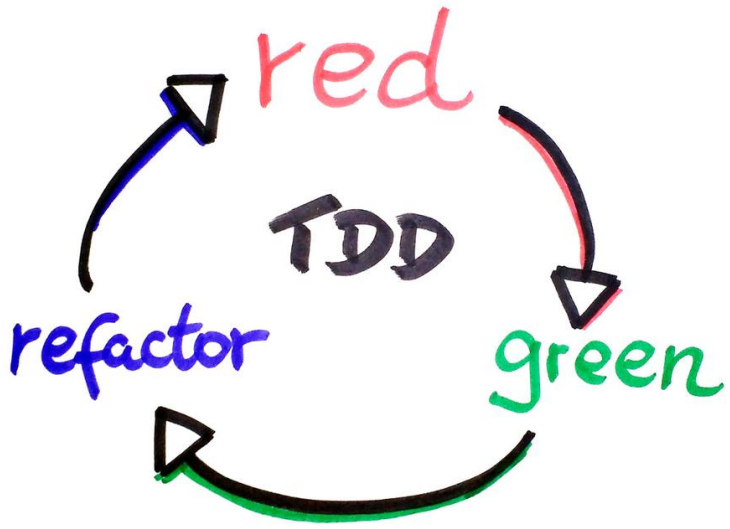
Test Driven Development (TDD)

- White Box Tests
- Write Tests before productive code
- Tests are the contracts
- Red-Green-Refactor Cycle
- Part of Extreme Programming (XP)

Red-Green-Refactor Cycle of TDD

Red-Green-Refactor Cycle:

- Add a test
- This test should fail in the beginning. (red)
- Write productive code that only turns the test to success. (green)
- Test and productive code should be refactored.



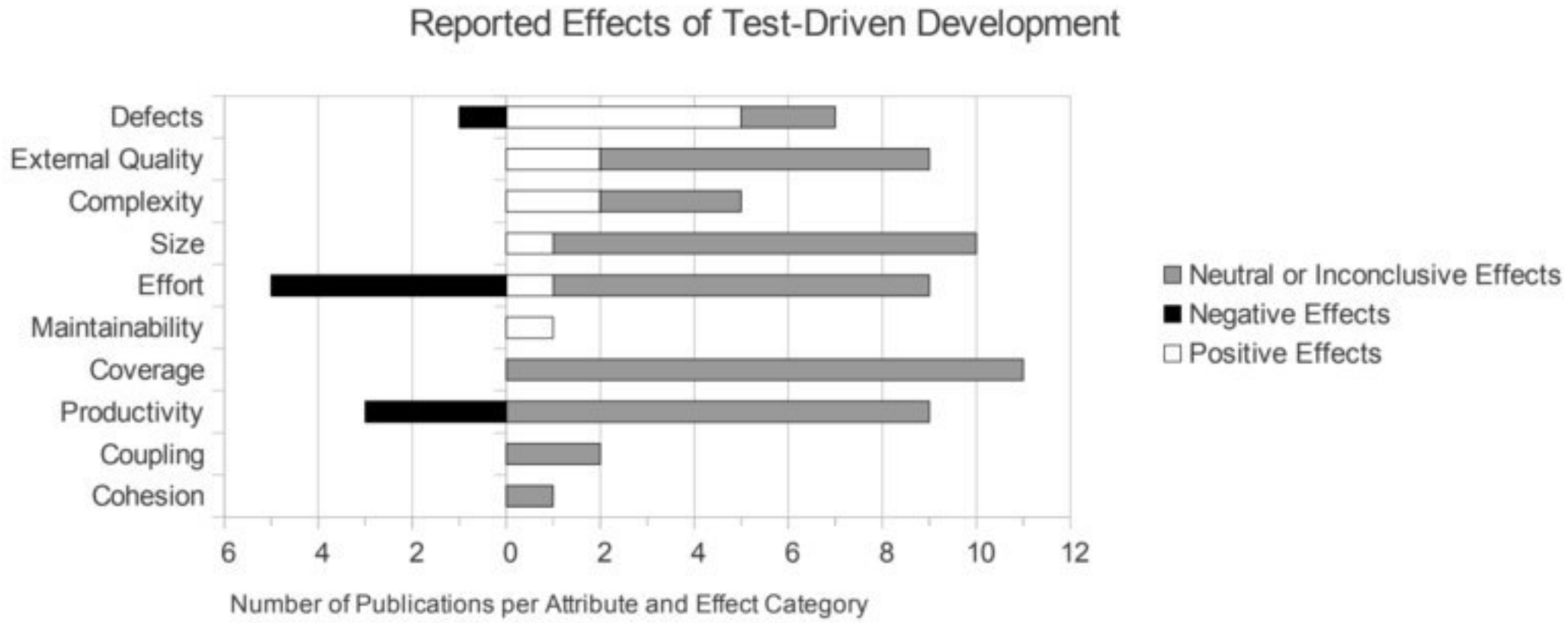
Pros of Test Driven Development (TDD)

- documentation for code
- Easier automation
- programmers really understand their code
- no untested code inside code base (more stability)
- no or less redundant code by just-in-time refactorization
- testing while writing forces to make interfaces clean enough to be tested
- clear and testable architecture by TDD as design strategy (forces good architecture; modular design)
- early warning to design problems

Cons of Test Driven Development (TDD)

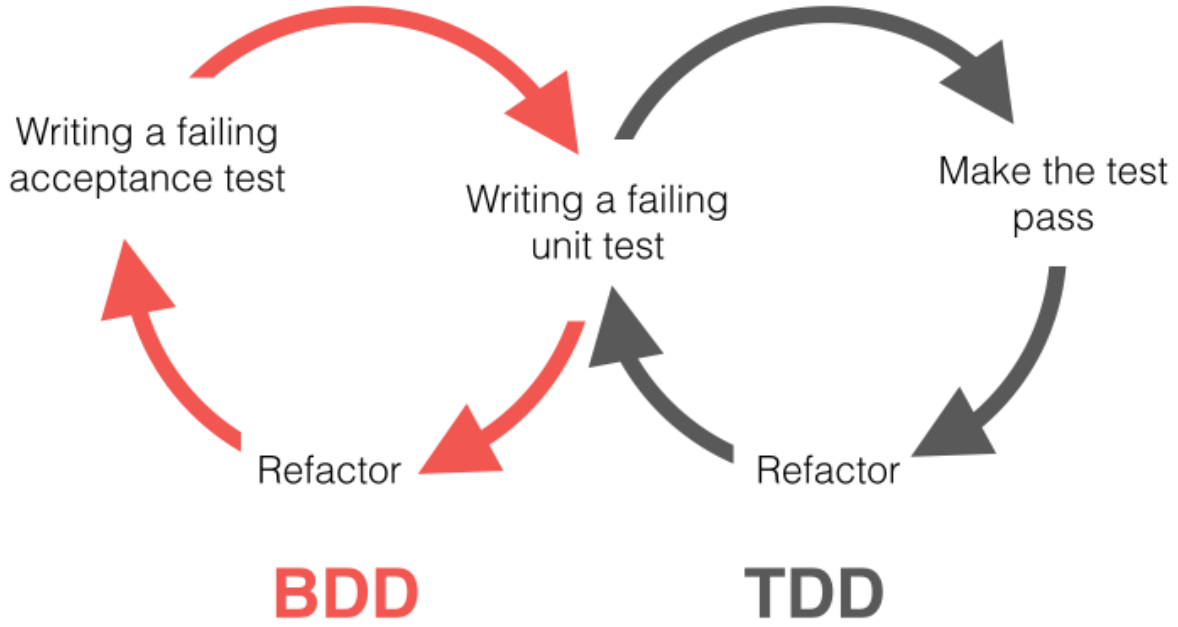
- in the beginning it takes more time to develop tests
- not so much flexible
- hard to apply to existing legacy code
- like any technique TDD can be carried to an extreme
- tests may be hard to write, for example beyond the unit testing level
- whole team has to write and maintain tests

TDD - reported project feedback



Behavior Driven Development (BDD)

- black-box testing
- emerged by Test Driven Development
- uses simple domain-specific language by natural language constructs



Behavior Driven Development (BDD)

- English like sentences
  - express behavior
  - expected outcome
- unit test names sentences with conditional verbs (should)
- acceptance tests
  - use standard agile framework of a user story
  - acceptance criteria
    - written of terms of scenarios
    - implemented as classes

Example of Behavior Driven Development (BDD)

- Behat PHP BDD Testing Framework
  - Feature inside .feature file as text format

***Feature**: Product basket*

In order to buy products

As a customer

I need to be able to put interesting products into a basket

Rules:

- VAT is 20%

- Delivery for basket under €10 is €3

***Scenario**: Buying a single product under €10*

***Given** there is a “Sith Lord Lightsaber”, which costs €5*

***When** I add the “Sith Lord Lightsaber” to the basket*

***Then** I should have 1 product in the basket*

***And** the overall basket price should be €9*

Example Behavior Driven Development (BDD)

```
class FeatureContext implements Context
{
  private $shelf;
  private $basket;

   public function __construct()
   {
     $this->shelf = new Shelf();
     $this->basket = new Basket();
   }
   /** @Given there is a :product, which costs €:price
   public function thereIsAWhichCostsPs($product, $price)
   {
    $this->shelf->setProductPrice($product,$price);
   }
   /** @When I add the :product to the basket **/
   public function iAddTheToTheBasket($product)
   {
      $this->basket->addProduct($product);
   }
   /** @Then I should have :count product in the basket **/
   public function iShouldHaveProductInTheBasket($count)
   {
     PHPUnit_Framework_Assert::assertCount(intval($count),$this->basket);
   }
   /** @Then the overall basket price should be €:price **/
   public function theOverallBasketPriceShouldBePs($price)
   {
    PHPUnit_Framework_Assert::assertSame(floatval($price), $this->basket->getTotalPrice());
   }
```

Pros of Behavior Driven Development (BDD)

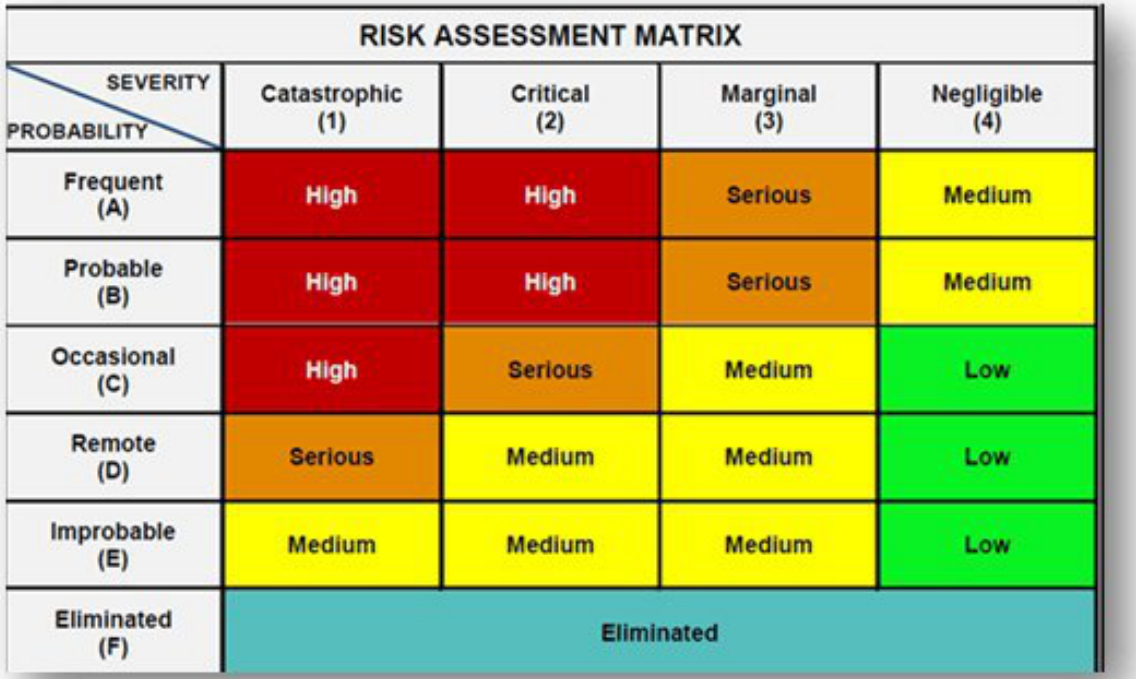
- agile workflow
- communication between users and developers
- short response time

Cons of Behavior Driven Development (BDD)

- if user not available, difficult to work with user stories
- need to dedicate a team of developers to work with users

Risk-based testing

- Use risk analysis to prioritize what should be tested
- “What happens if this class or function fails? How probable?”
- Severity + probability
- “Risk coverage” instead of “code coverage” as a metric



RISK ASSESSMENT MATRIX				
SEVERITY / PROBABILITY	Catastrophic (1)	Critical (2)	Marginal (3)	Negligible (4)
Frequent (A)	High	High	Serious	Medium
Probable (B)	High	High	Serious	Medium
Occasional (C)	High	Serious	Medium	Low
Remote (D)	Serious	Medium	Medium	Low
Improbable (E)	Medium	Medium	Medium	Low
Eliminated (F)	Eliminated			

Risk-based testing - risks in LimeSurvey

1. Catastrophic: Data loss or security issues
2. Critical: Blocking issues, e.g. “Next” button doesn’t work
3. Marginal: Some rarely used option won’t save
4. Negligible: Cosmetic issues, spelling etc

“Negligible” can still have high priority if frequency is high.

Unit Tests vs Integration Tests vs UI Tests

- Unit Test: only testing one unit = class or function (helpers…)
- Integration Test: testing interaction between classes
- UI Test: black box test for UI
- Sanity test: Test basic functionality of a feature, no edge cases

Good tests

- Tests only one thing (one assert per test, ultimately)
- Order of tests must not matter
- Independent tests
- Proper clean-up (tear-down)

Bad tests

- Tests multiple things
- Hard to read
- Hard to maintain (tests superficial or cosmetic things)
- Tests which are affecting other tests after (session, $_POST, …)
- Worse test is the test that doesn’t exist?

Negative tests

- Test proper *failure*
- Important to make sure silent failures never happen
- Example: throw invalid argument exception, assert() in code

Testable code

Testable function:

- Clear relation between input and output
- Short
- No side-effects (but uncommon in data-driven application like LS)
  - Loading or saving data
  - Reading or writing to files
  - Reading or writing global state

Testable function - getGidPrevious()

```
function getGidPrevious($surveyid, $gid)
{
   $surveyid = (int) $surveyid;
   $s_lang = Survey::model()->findByPk($surveyid)->language;
   $qresult = QuestionGroup::model()->findAllByAttributes(array('sid' => $surveyid, 'language' => $s_lang),
array('order'=>'group_order'));
   $i = 0;
   $iPrev = -1;
   foreach ($qresult as $qrow) {
      $qrow = $qrow->attributes;
      if ($gid == $qrow['gid']) {$iPrev = $i - 1; }
      $i += 1;
   }

   if ($iPrev >= 0) {$GidPrev = $qresult[$iPrev]->gid; } else {$GidPrev = ""; }
   return $GidPrev;
}
```

- Side-effects - boo
- Requires database setup to test

Testable function - getGidPrevious() improved

```
function getGidPrevious($questionGroups, $gid)
{
   $i = 0;
   $iPrev = -1;
   foreach ($questionGroups as $qrow) {
      $qrow = $qrow->attributes;
      if ($gid == $qrow['gid']) {$iPrev = $i - 1; }
      $i += 1;
   }

   if ($iPrev >= 0) {$GidPrev = $questionGroups[$iPrev]->gid; } else {$GidPrev = ""; }
   return $GidPrev;
}
```

- No side-effects
- Side-effects moved higher up in stacktrace (saving and loading data, e.g.)
- Does not require setup or tear-down

Testable classes

Dependency injection (DI)

Example: class A depends on class B, want to test A without testing B

Example in LS: Almost all classes depend on Permission model - how to ignore it in tests?

Testing without Permission model

- Now: Use `\Yii::app()->session['loginID'] = 1;` (superadmin)
- Bad: Not contained, affects tests run after (or needs tear-down)
- Future: Inject mock Permission model which always returns true
- Use PermissionInterface instead of Permission model
- Related: PHP 7 anonymous classes (on-the-fly classes)

Without dependency injection

```
// An example without dependency injection
public class Client {
    // Internal reference to the service used by this client
    private ExampleService service;

    // Constructor
    Client() {
        // Specify a specific implementation in the constructor instead of using dependency injection
        service = new ExampleService();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

With constructor dependency injection

```
// An example with constructor dependency injection
public class Client {
    // Internal reference to the service used by this client
    private ExampleService service;

    // Constructor
    Client(Service service) {
        // Save the reference to the passed-in service inside this client
        this.service = service;
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

Dependency injection in PHP

PSR-11 supported by Yii 3

Container Interface (https://www.php-fig.org/psr/psr-11/)

Our testing system

- PHPUnit
- Selenium
- Geckodriver
- Firefox
- Facebook WebDriver
- Jest (JavaScript) Framework
- Travis (Docker)

Folder structure

- unit
  without selenium
- functional
  with selenium

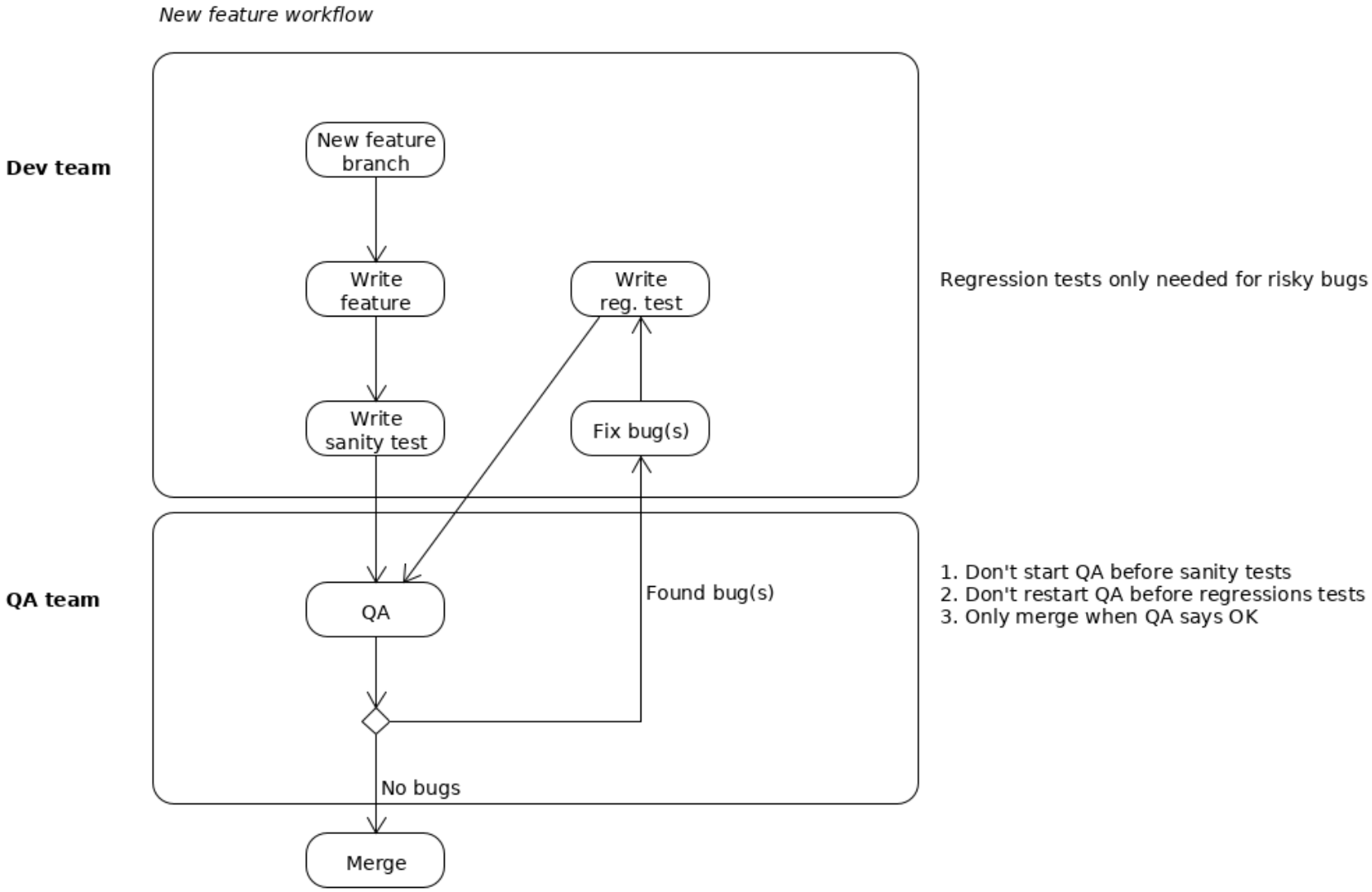
Example unit test

```
public function testToken()
{
  // Get our token.
  $tokens = \TokenDynamic::model(self::$surveyId)->findAll();
  $token = $tokens[0];

  // Change lastname.
  $token->lastname = 'last';
  $token->encryptSave();

  // Load token and decrypt.
  $tokens = \TokenDynamic::model(self::$surveyId)->findAll();
  $token = $tokens[0];
  $token->decrypt();

  $this->assertEquals('last', $token->lastname);
}
```

Example functional test

```
class LanguageChangerTest extends TestBaseClassWeb
{
	public function testLanguageSelect()
	{
	$web = self::$WebDriver;
	$url = $this->getSurveyUrl('pt');

	$web->get($url);
	$web->changeLanguageSelect('de');

	sleep(1);

	$text = $web->findByCSS('.question-count-text');

	$this->assertContains($text->getText(), 'In dieser Umfrage sind 2 Fragen enthalten.');
	}
}
```

Possible workflow for new feature



https://bugs.limesurvey.org/view.php?id=15336