[Unit and Integration Testing Guidelines](https://wiki.eisgroup.com/display/CRC/Unit+and+Integration+Testing+Guidelines)

# Anti-patterns

## **1- PowerMock intensive usage**

Intensive usage of  PowerMock leads to the following issues:

1. Introducing high coupling between classes and as a result significantly increased cost of any change that affects the static class, because there is no way to localize ripple effect of the change.
2. Slow unit tests that cannot be used by developers actively for verification of their changes. Nobody wants to wait for UTs results for several hours.
3. Fragile unit tests. With high coupling, it is hard to test units in isolation. Such UTs are hard to maintain.

To avoid all those issues the following rules need to be followed by developers and verified by reviewers:

1. PowerMock can be used only to mock external libs. It is strictly forbidden to use PowerMock to stub project classes.

## **2 - Creating \*MockImpl classes**

**Problem statement**: \*MockImpl classes are created instead of using Mockito.

**Consequences:**Complex unit tests which are hard to maintain.

**The solution**: Use Mockito to mock dependencies.

## **3 - Using JUnit 3 and JUnit 4 approaches simultaneously**

**Problem statement**: Test class extends TestCase and uses JUnit 4 annotations.

**Consequences:** Annotations are ignored.

**The solution**:  Use either JUnit 3 or JUnit 4 approach.

Please see the details in the [Differences Between JUnit3 and JUnit4](https://wiki.eisgroup.com/display/CRC/Differences+Between+JUnit3+and+JUnit4)

## **4 - Using @Mock(answer=Answers.RETURNS\_DEEP\_STUBS) for non-legacy code**

**Problem statement**: Annotation @Mock(answer=Answers.RETURNS\_DEEP\_STUBS) is used for non-legacy code.

**Consequences:**With such answer type Mockito generates stubs for deep graph of objects and it exhaust PermGen space. And when you try to do this for large graph like ABCAutoPolicyEntity, we will have OutOfMemoryError in most cases. Also it isn’t recommended answer type according to Mockito documentation.

“WARNING: This feature should rarely be required for regular clean code! Leave it for legacy code. Mocking a mock to return a mock, to return a mock, (...), to return something meaningful hints at violation of Law of Demeter or mocking a value object (a well known anti-pattern).”

**The solution**:

1. Refactor your production code to avoid violations of Law of Demeter.
2. Do not mock value objects (DTO, entities, etc).

## **5 - Mocking value object**

**Problem statement**: Test mocks classes that doesn’t have any behavior (mocking state instead of behavior).

**Consequences:**

1. Overhead for creating an interface/implementation pair just to control which time values are returned.
2. Memory overhead.

Typical example of this anti-pattern:

|  |
| --- |
| @Mock  ABCAutoPolicyEntity policySummary;    @Before  **public** **void** init() {      MockitoAnnotations.initMocks(**this**);      ...  }    @Test  **public** **void** someTest() {      ...      when(policySummary.getPolicyDetail()).thenReturn(policyDetail);      ...  } |

**The solution**:   Create instances with the appropriate values and use them. For instance, bad example from previous section can be replaced by the following code:

|  |
| --- |
| ABCAutoPolicyEntity policySummary;    @Before  **public** **void** init() {      MockitoAnnotations.initMocks(**this**);        policySummary = **new** ABCAutoPolicyEntity();  }    @Test  **public** **void** someTest() {      ...      policySummary.setPolicyDetail(policyDetail);      ...  } |