# Interaktionen in Unit Tests

## Definitionen

|  |  |
| --- | --- |
| Begriff | Definition |
| SUT (auch CUT) | System/Component Under Test (bei Unit Tests i.A. die zu testende Klasse rsp. eine Methode oder ein Zusammenspiel von Methoden) |
| DOC (auch collaborator) | Depends On Component (Komponenten/weitere Klassen von denen die SUT abhängt) |
| Worker | Klassen, deren Methoden die eigentliche Arbeit machen |
| Manager | Klasse, deren Methoden die Arbeit von worker-Klassen koordiniert. |
| Direct input | Methoden-Aufrufe (und deren Parameter) der SUT |
| Direct output | Rückgabewerte der SUT an die Test-Methode |
| indirect input | Rückgabe eines DOCs (collaborators) innerhalb einer SUT-Methode |
| indirect output | Argumente, die die SUT innerhalb einer ihrer Methoden an collaborators weitergibt |
| State testing | testet worker; benutzt nur direct input /output |
| interaction testing | Betrachtet, wie collaborators eines managers interagieren und wie dabei indirect input /output zwischen ihnen ausgetauscht wird. |

## Übung

Gegeben: package **ch02.x01Interactions**

Betrachten Sie den Code und füllen Sie folgende Tabelle zu Ende aus:

|  |  |
| --- | --- |
| Begriff | Beispiel im Code |
| SUT | FinancialService. calculateBonus() |
| DOC (collaborator) | clientDAO, calculator |
| Manager | FinancialService |
| Worker | Calculator (ClientDAO eher nicht, da keine echte Logik) |
| Direct input | ClientId, payment |
| Direct output | bonus (return) |
| indirect input | ClientType (clientDAO), bonus(calculator) |
| indirect output | ClientId (clientDAO), clientType & payment (calculator) |

## Braucht man wirklich interaction testing?

Con (which is a fallacy, though): Interaction Testing bricht Encapsulation und Information Hiding auf. Eine gut designte API sollte nur über ihre Schnittstellen testbar sein

Pro: Manche Funktionalität kann nur über Interaction Testing getestet werden.

Beispiel: Ein Cache

|  |
| --- |
| There are two storage locations, the "real one", with vast capacity and average access time, and the "cache", which has much smaller capacity but much faster access time. Let us now define a few requirements for a system with a cache.  When asked for an object with key X , our system with its cache should act according to the following simple rules:  1. if the object with key X is not in any storage location, the system will return null  2. if the object with key X exists in any storage location, it will be returned  a. if it exists in the cache storage, it will be returned from this storage location  b. the main storage location will be searched only if the object with key X does not exist in the cache storage  The point is, of course, to have a smart caching strategy that will increase the cache hit ratio.  With state testing we can only test two of them, 1 and 2. This is because state testing respects objects’ privacy. It does not allow one to see what the object is doing internally – something which, in our case, means that it cannot verify from which storage area the requested object has been retrieved. Thus, requirements 2a and 2b cannot be  verified using state testing.  (…)  Similar issues arise generally when testing managers which coordinate the efforts of others. |

Ein guter Teil dessen, was die Funktionalität von **Mockito** ausmacht, beschäftigt sich mit **Interaction Testing**.