Methodology

Bij object-georiënteerd ontwerp is het de kunst om de juiste verantwoordelijkheid bij de juiste objecten te leggen en zo een helder ontwerp met losse koppeling en hoge cohesie te bouwen. Test Driven Development (TDD) stuurt je die richting op, maar nog niet helemaal. Het helpt om los gekoppelde objecten te krijgen, aangezien te veel koppelingen het test-codeer-refactor-ritme in de weg zitten.
Responsibility Driven Design is een aanpak die nog een stap verder gaat. De focus verschuift van de toestand van objecten naar interacties en verantwoordelijkheden. In dit artikel laten we zien hoe responsibility driven design helpt om een OO ontwerp met hoge cohesie en lage koppeling te krijgen en hoe test driven development met mock objects dat faciliteert.

Responsibility Driven Design met Mock Objects

Marc Evers
werkt als zelfstandig coach, trainer en adviseur op het gebied van (agile) softwareontwikkeling. Hij is te bereiken op marc@piecemealgrowth.nl

Rob Westgeest
heeft jarenlange ervaring in OO Software ontwikkeling. Hij is te bereiken op rob@westgeest-consultancy.com

Verantwoordelijkheden zijn het fundament van een object-georiënteerd systeem. Een OO systeem kun je voorstellen als een kooi waarin objectjes leven. Als de buitenwereld een opdracht geeft aan het systeem, werken de objecten samen om die opdracht uit te voeren. Het fundament van die samenwerking is *verantwoordelijkheid*: elk object *doet* wat hij het beste kan *doen* en delegeert de rest naar zijn buurman. De focus op wat een object kan doen ligt centraal in responsibility driven design (Wirfs-Brock & McKean, 2003).
De kunst is om verantwoordelijkheden dicht bij de kennis die daarvoor nodig is te leggen. Zo krijg je kleine, begrijpelijke objecten die een losse koppeling hebben met hun omgeving.
Een veel gebruikte techniek die responsibility driven design ondersteunt, is de *CRC kaarten* techniek (*C*lass, *R*esponsibility, *C*ollaboration) (Beck & Cunningham, 1989). Elk kaartje representeert een klasse en geeft weer welke verantwoordelijkheden de klasse heeft en met welke andere objecten (*collaborators*) hij samenwerkt om die verantwoordelijkheden te vervullen.

Test Driven Development

Test Driven Development (TDD) is een ontwikkeltechniek die responsibility driven design goed ondersteunt. TDD is een techniek om een ontwerp te laten ontstaan vanuit het schrijven van tests. Hierbij onderhoud je de code continu (*refactoring*) zodat het ontwerp optimaal blijft passen op de tot dan toe geschreven en werkende tests (Beck, 2003). De stappen zijn bedrieglijk simpel:
schrijf eerst een (unit) test
laat de test falen
voeg stap voor stap code toe om te test te laten slagen
refactor de code – verwijder duplicatie en laat de code voor zichzelf spreken
herhaal
In de praktijk is dit nog helemaal niet zo eenvoudig. TDD stuurt je tot het identificeren van zo onafhankelijk mogelijke eenheden en dwingt een ontwerp af met hoge cohesie en lage koppeling (*high cohension/low coupling*): werkelijk alle afhankelijkheden zitten je in de weg.

Mock objecten en interaction based testing

Velen die beginnen met TDD worstelen ermee om enerzijds alle afhankelijkheden goed onder controle te krijgen en anderzijds toch de interne toestand van objecten te kunnen opvragen om te controleren of een bepaalde methode-aanroep het juiste effect heeft.
Tim Mackinnon en zijn collega's (Engelse pioniers op het gebied van agile softwareontwikkeling)

worstelden hier ook mee in 1999. Zij voegden vaak extra getters toe om de toestand van objecten te kunnen valideren. Hun baas John Nolan was niet gecharmeerd van dat breken van encapsulatie: *"I want no getters in my code!"* (MacKinnon et al., 2000) (Freeman et al., 2004)
Tim en zijn collega's kwamen op het idee om zich te richten op interacties. Zij maakten een soort stubs die collaborators vervangen en waar je de te verwachten methode-aanroepen kon specificeren. Deze bijzondere stubs noemden ze *Mock Objects* (kortweg *mocks*). De oorspronkelijke ideeën zijn in de loop der tijd verfijnd en tegenwoordig zijn er mock object frameworks voor o.a. Java, C#, Python, Ruby en C++. Bekende frameworks voor Java zijn jMock (www.jmock.org) en EasyMock (www.easymock.org).
De stijl van testen die hier uit voortvloeit heet *interaction based testing* (Fowler, 2004). Interaction based testing richt zich, net als responsibility driven design, op verantwoordelijkheden van en interacties tussen objecten, en niet op toestand.

Een voorbeeld

We lichten responsibility driven design met mock objects toe aan de hand van een voorbeeld. We ontwerpen een eenvoudige workflow engine. Het initiele ontwerp met CRC kaarten ziet er als volgt uit:

Een WorkflowProcess zorgt dat alle taken worden uitgevoerd op een WorkItem. Hiervoor werkt hij samen met Task en WorkItem. Een Task zorgt ervoor dat zijn eigenaar zijn werk uitvoert op WorkItems en werkt samen met Owners en WorkItems.

In onze eerste test verifiëren we dat wanneer een WorkflowProcess wordt gestart, de eerste taak wordt gestart. Om onze verwachtingen rond de interacties te specificeren en verifiëren, gebruiken we het Mockito framework (www.mockito.org):

```
import org.junit.Test;
import static org.mockito.Mockito.*;

public class WorkFlowProcessTest {
  @Test
  public void startingAProcessShouldInitiateATask
() {
        Task task = mock(Task.class);
        WorkItem workItem = mock(WorkItem.class);
        WorkflowProcess    process    =    new
WorkflowProcess(task);

        process.start(workItem);

        verify(task).start(workItem);
  }
}
```

We instantiëren het object onder test (WorkflowProcess) en maken mocks voor de collaborators *task* en *workitem*. Vervolgens komt de kern van de test – het testen van het gedrag en de interacties van de *start* methode van WorkflowProcess. Tot slot verifiëren we onze verwachtingen rond de interacties: de *start* methode moet aangeroepen zijn op *task*, met als argument *workitem*. De methoden *mock* en *verify* zijn static methoden op de org.mockito.Mockito klasse.
We laten ons leiden door de compiler en maken interfaces voor Task en WorkItem, een klasse WorkflowProcess en de benodigde methoden en constructors. We laten deze methoden in eerste instantie leeg: we willen onze tests de implementatie laten sturen en willen de test eerst zien falen:

```
  org.mockito.exceptions.verification.
WantedButNotInvoked:
  Wanted but not invoked:
  task.start(
      Mock for WorkItem, hashCode: 6613606
  );
```

Met andere woorden, de aanroep van de *start* methode op *task* werd wel verwacht maar is niet uitgevoerd.
De volgende implementatie laat de test slagen:

```
public class WorkflowProcess {
  private final Task task;

  public WorkflowProcess(Task task) {
        this.task = task;
  }

  public void start(WorkItem workItem) {
        task.start(workItem);
  }
}
```

Tell, Don't Ask

De test maakt de verwachtingen rond de interacties tussen een WorkflowProcess object en zijn collaborator *task* duidelijk. Deze stijl van testen stuurt in de richting van een *tell, don't ask* stijl van ontwerpen: in plaats van allerlei data uit een object te halen met behulp van getters (*ask*) en vervolgens daarmee iets te doen, geef je een object opdracht om het zelf te doen (*tell*), waarbij het object bepaalt hoe hij het doet.
In plaats van zogenaamde *trainwreck*-constructies als:

```
  process.getFirstTask().getOwner().
workOn(workItem);
```

benoem je het gedrag expliciet in methoden:

```
  process.start(workItem);
```

Een test met mocks voor het bovengenoemde trainwreck zou er ongeveer zo uitzien:

```
public class WorkflowProcessTest {
  @Test
  public void startingAProcessShouldInitiateATask
() {
        Person owner = new Person();
        Task task = mock(Task.class);
        WorkItem workItem = mock(WorkItem.class);
```

```
        WorkflowProcess process = new
WorkflowProcess(task);

        stub(task.getOwner()).toReturn(owner);

        process.getFirstTask().getOwner().
workOn(workItem);

        verify(task).getOwner();
        assertEquals(workItem, owner.getCurrentWor-
kItem());
  }
}
```

Zo'n test verliest zijn focus. Testen we nu het starten van een proces of juist *getFirstTask* of *getOwner* of *workOn* of allemaal? De aanwezigheid van veel afhankelijkheden maakt de test niet alleen lastiger op te zetten, maar ook te lezen en te onderhouden. Daarom sturen testen met mocks je naar een responsibility driven ontwerp.

Rollen en concepten

Tijdens het ontwikkelen vind je op deze manier ook nieuwe concepten. Voor concepten definieren we rollen, gerepresenteerd door Java interfaces. Het gebruik van rollen en interfaces zorgt tegelijkertijd voor losse koppeling omdat objecten alleen afhankelijk zijn van abstracte interfaces en niet van concrete implementatieklassen.

Laten we naar de volgende test kijken. De taak is geïdentificeerd als concept. Het is echter geen klasse maar een interface. Wie implementeert die interface? *TaskImpl* ligt voor de hand. We vinden dat echter een mindere keuze, die we gebruiken als we niets beters weten te verzinnen. Dat is tevens het moment dat we nog even nadenken of we wel een verschil willen maken tussen interface en implementatie. *SimpleTask* is een betere kandidaat. We kunnen ons voorstellen dat er ook complexe parallelle of sequentiële taken zijn, al willen we daar niet te veel op vooruitlopen.

Als een taak wordt gestart zal hij zijn eigenaar vragen om aan het WorkItem te werken. Hebben we een *Person* nodig als eigenaar? Mogelijk, maar waarom niet *TaskOwner*, de rol die we net noemden? Hier is de test:

```
@Test
public void startShouldMakeOwnerWorkOnWorkItem(){
  TaskOwner owner = mock(TaskOwner.class);
  WorkItem workItem = mock(WorkItem.class);
  SimpleTask task = new SimpleTask(owner);

  task.start(workItem);

  verify(owner).workOn(workItem);
}
```

Hier specificeren we dat owner de boodschap *workOn(workItem)* ontvangt wanneer een taak van de betreffende eigenaar wordt gestart.

Wie doet wat?

We laten de TaskOwner even voor wat die is en gaan door met SimpleTask. Wat nu als de taak afgerond is? In dat geval moet de volgende taak gestart worden. Bij wie hoort deze verantwoordelijkheid? Bij SimpleTask, bij TaskOwner of bij WorkflowProcess? We zien ontwikkelaars vaak die verantwoordelijkheid neerleggen daar waar de situatie ontstaat, bijvoorbeeld:

```
public class SimpleTask implements Task {
  WorkflowProcess process
  ...
  public void done(){
        Task next = process.getNextTask();
        next.start(workItem);
  }
  ...
}
```

Dit ziet er simpel uit en werkt op zich wel. Maar wie beheert de taken in deze implementatie? WorkflowProcess? SimpleTask? Het probleem is dat ze het beide een beetje doen. Onze CRC kaartjes laten zien dat WorkflowProcess als verantwoordelijkheid heeft: *guide work items through tasks.* Het starten van een volgende taak hoort daar volgens ons bij. Dit leidt ons naar de volgende test:

```
public class SimpleTaskTest {
  @Test
  public void doneShouldInformTheProcess(){
        TaskOwner owner = mock(TaskOwner.class);
        TaskParent parent = mock(TaskParent.
class);
        SimpleTask task = new SimpleTask(parent,
owner);

        task.done();

        verify(parent).taskDone(task);
  }
}
```

We verwachten dat niet WorkflowProcess maar TaskParent het bericht *taskDone* ontvangt. Dat is een nieuw concept (een nieuwe Java Interface). De klasse WorkflowProcess zal *taskDone* implementeren, voor de rol TaskParent maken we een mock. De implementatie ziet er dan als volgt uit:

```
public class SimpleTask implements Task {
  private TaskParent parent;
  ...
  public void done() {
        parent.taskDone(this);
  }
}
```

SimpleTask heeft dus geen afhankelijkheid naar de concrete klasse WorkflowProcess maar naar de rol TaskParent.
Vervolgens implementeren we *taskDone* in WorkflowProcess met de verwachting dat dan de volgende taak gestart wordt.

```
public class WorkflowProcessTest {
  @Test
  public void endingFirstTaskShouldStartSecond() {
        Task task1 = mock(Task.class);
        Task task2 = mock(Task.class);
        WorkItem workItem = mock(WorkItem.class);
        WorkflowProcess process = new
WorkflowProcess(task1, task2);

        process.start(workItem);
        process.taskDone(task1);

        verify(task2).start(workItem);
  }
}
```

met bijbehorende implementatie:

```
public void taskDone(Task task) {
  successorOf(task).start(workItem);
}
```

Enkele vuistregels

Unit testen is moeilijk. Het vergt vaardigheid en oefening. Interaction based testing is ontstaan uit een worsteling van anderen en kan je helpen om je ontwerp te verbeteren. Het stuurt je naar losser gekoppelde, meer coherente objecten, met de focus op gedrag in plaats van toestand.
Interaction based testing vergt ook oefening, wellicht nog meer dan 'traditionele' unit tests. Mocks worden vaak verkeerd geïnterpreteerd en verkeerd gebruikt. We geven hier enkele vuistregels voor het gebruik van mocks.

Mock rollen en niet objecten

Hoewel de meeste mock object frameworks als jMock, EasyMock en Mockito concrete klassen kunnen mocken, denken wij dat juist de focus op interfaces/rollen iets toevoegt. Het houdt de aandacht op het object onder test, de rollen die dit object vervult en de verantwoordelijkheden die het delegeert naar de samenwerkende rollen – precies datgene waar responsibility driven design om draait.

Gebruik mocks voor opdrachten en stubs voor getters

Getters breken encapsulatie. We krijgen geregeld te horen dat dat niet per se het geval is: *getXYZ*() kan een property van een object zijn, maar zegt niets over hoe deze geïmplementeerd is; *XYZ* kan wel berekend worden uit allerlei andere velden. Dat is waar, maar *getXYZ*() communiceert in het ontwerp dat andere objecten gebruik kunnen maken van property *XYZ*. Sterker nog, de getter *getXYZ*() ontstaat vaak omdat een ander object *XYZ* nodig heeft voor bepaald gedrag en dat is nu net de kern van het breken van encapsulatie. Stel jezelf hier de vraag: zit property XYZ op de verkeerde plaats of leg ik het gedrag op de verkeerde plaats?

Mocking is ontstaan vanuit de gedachte "*ik wil geen getters in mijn code!*". Mocks zijn bedoelt om opdrachten naar aanpalende rollen te verifiëren. Wanneer een object een getter van een ander object nodig heeft – ja we weten het: liever niet – mock die getters dan liever niet. Wanneer simpelweg een waarde nodig is, is simpel stubben voldoende en beter leesbaar.

Mock niet alle aanroepen die je in je implementatie doet

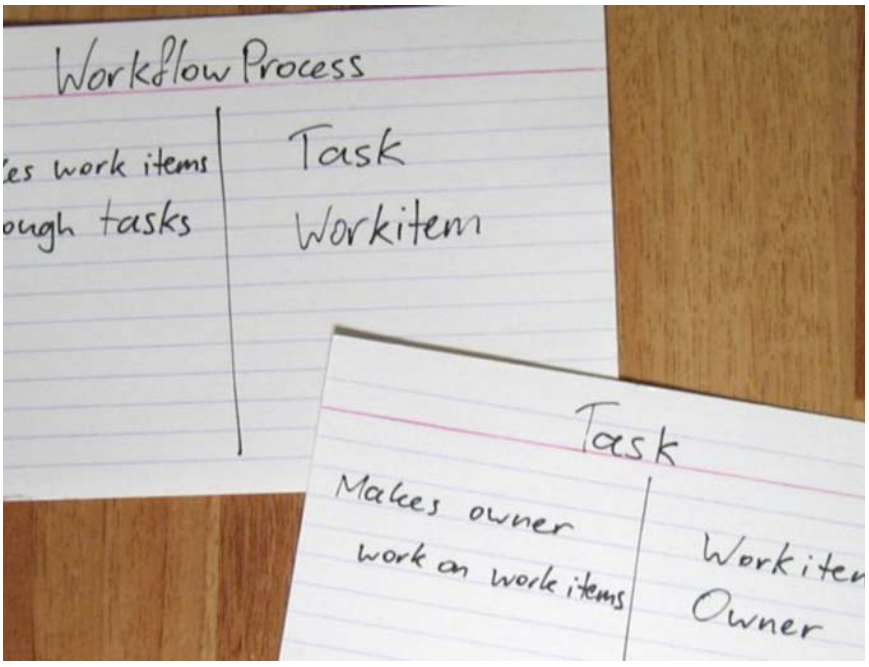
Een risico van TDD en van mocks in het bijzonder is dat tests te fijnmazig het gedrag specificeren. Zodra in de test een kopie van de implementatie van de methoden ontstaat, kan er welhaast niets worden veranderd zonder de test te breken. De kunst is het vinden van een goede balans tussen voldoende specificeren van de interactie tussen objecten en voldoende ruimte laten aan de implementatie om te wijzigen zonder dat de test breekt.

Houd je mocks en echte objecten synchroon

Als je tests met mocks gebruikt, bevatten je tests aannames hoe bepaalde rollen zich zouden moeten gedragen. Je moet er wel voor zorgen dat deze aannames kloppen met het echte gedrag. Het is mogelijk dat de verwachtingen en het echte gedrag compleet uiteenlopen: alle unit tests slagen in dat geval, maar het systeem als geheel werkt niet.

Voeg geen gedrag toe aan mocks

Gedrag toevoegen aan mocks of stubs maakt tests onleesbaar en is in onze ervaring niet nodig. Ontwikkelaars laten soms mocks andere mocks



retourneren die weer mocks retourneren. Zo bouw je trainwrecks in je verwachtingen. Soms worden zelfgebouwde mocks gebruikt met complex gedrag. De tests testen dan niet alleen het object onder test, maar ook de zelfgebouwde mocks.

Mock alleen je eigen klassen

Mock alleen klassen waar je controle over hebt. Daar kun je interfaces uit extraheren om mocks voor te maken. Zelfs als een 3rd party framework of bibliotheek interfaces aanbiedt, is het een goed idee om mocks te maken op basis van eigen interfaces. Het gaat om het mocken van rollen die je zelf identificeert.

Conclusie

Mock object frameworks als jMock, EasyMock en Mockito zijn bruikbare hulpmiddelen die je helpen richten op het gedrag van en interacties tussen objecten in een object-georiënteerd systeem. De stijl van testen die daaruit voortvloeit heet interaction based testing en stuurt je naar een responsibility driven design. Het helpt je om een ontwerp te laten groeien bestaande uit kleine, heldere, autonome objecten die losgekoppeld zijn van elkaar.

Hoeveel het je ook kan helpen, het is en blijft een vaardigheid. Het is gemakkelijk om onleesbare en ononderhoudbare tests en code te maken. Vaardigheden kun je verbeteren door oefening met medebeoefenaars. Een praktijk die in de laatste jaren opkomt is de coding dojo (www.coding-dojo.org), naar analogie van dojo's in oosterse vechtsporten. In een coding dojo oefen je samen met collega's om je (test driven) codeervaardigheden te verbeteren. «

Referenties

Rebecca Wirfs-Brock & Alan McKean, *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley 2003

Kent Beck, Ward Cunningham, *A Laboratory For Teaching Object-Oriented Thinking*, 1989

Kent Beck, *Test Driven Development: By Example*, Addison-Wesley 2002

Martin Fowler, *Mocks Aren't Stubs*, 2004

Tim Mackinnon, Steve Freeman, Philip Craig, *Endo-Testing: Unit Testing with Mock Objects*, 2000

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes, *Mock Roles, not Objects*, 2004