API-ontwerp en encapsulatie van collecties

Bart Jacobs

22 februari 2015

Een belangrijke vaardigheid is het ontwerpen van APIs (Application Programming Interfaces, m.a.w. de verzameling van types (klassen en interfaces) en methodes die een bibliotheek ter beschikking stelt van haar cliënten). In het bijzonder is het voor het project belangrijk de API van de domeinlaag, zoals die gebruikt wordt door de presentatielaag, goed te ontwerpen.

Een goed ontworpen API betekent in de eerste plaats een goed gedefinieerde API. Een API is goed gedefinieerd als zowel haar syntax als haar semantiek goed gedefinieerd is.

Het goed definiëren van de syntax van een API De syntax van een API is goed gedefinieerd als het duidelijk is welke types en methodes behoren tot de API, en welke niet. Als het een API betreft van een bibliotheek die slechts bestaat uit één Java-package, dan kan dit gemakkelijk gerealiseerd worden door de types en methodes die behoren tot de API te declareren als publiek, en de andere als package-accessible. Echter, als de bibliotheek bestaat uit meerdere packages, en deze packages moeten meer toegang hebben tot elkaar dan een externe gebruiker, dan is het soms nodig types en methodes als publiek te declareren, terwijl ze toch niet tot de API van de bibliotheek behoren. In dat geval is het nodig op een andere manier aan te duiden, welke types en methodes behoren tot de API en welke niet. Een voor de hand liggende manier is dat dit vermeld wordt in de Javadoc voor de bibliotheek. Een iets systematischere manier is het gebruik van een Java-annotatie. Het pakket capsules [1] definieert een specifieke manier om de API van een bibliotheek bestaande uit meerdere packages te definiëren, en biedt ook tools om Javadoc te genereren voor dergelijke bibliotheken, alsook om statisch na te gaan dat cliëntcode enkel gebruik maakt van de types en methodes die behoren tot de API.

Het goed definiëren van de semantiek van een API Een goed gedefinieerde API vereist echter niet alleen dat de syntax goed gedefinieerd is, maar ook de semantiek. Met andere woorden: het moet duidelijk zijn wat het precieze gedrag is van elke methode en constructor van de API. Idealiter is het gedrag van een methode of constructor onmiddellijk duidelijk dankzij een goede keuze van methodenaam, parameternamen, parametertypes, en resultaattype. Zoniet, dan moet Javadoc-documentatie voorzien worden.

Een specifieke vaak voorkomende bron van onduidelijkheid wat betreft de semantiek van APIs zijn collecties: als een methode een collectietype heeft als parametertype of als resultaattype, is het vaak niet duidelijk wat het precieze gedrag is van de bibliotheek wat betreft deze collecties.

Hierbij moeten we onderscheid maken tussen twee soorten collectietypes: wijzigbare (mutable) collectietypes en onwijzigbare (immutable) collectietypes. Voor onwijzigbare collectietypes geldt dat de toestand van een instantie van een dergelijk type na constructie nooit verandert; als het mogelijk is dat van een instantie van een collectietype de toestand na constructie verandert, spreken we van een wijzigbaar collectietype. Merk op dat alle types in java.util, zoals List en ArrayList, wijzigbare collectietypes zijn. (Je kan weliswaar via Collections.unmodifiableList een object creëren waarvan de mutator-methodes een exception gooien, en je kan zelfs een klasse definiëren die interface List implementeert en waarvan de instanties onwijzigbaar zijn, maar het punt is dat je louter uit het feit dat een object van het type List is, niet kunt afleiden of je de toestand van dit object kunt wijzigen of niet.)

Wijzigbare collectietypes in APIs Wanneer een API-methode een parameter heeft waarvan het type een wijzigbaar collectietype is, is het niet onmiddellijk duidelijk wat het gedrag van de methode is. In het bijzonder is het niet duidelijk of het toegelaten is dat de cliënt, nadat de API-methode terugkeert, nog wijzigingen aanbrengt in de collectie, en zo ja, wat het effect is van deze wijzigingen op de toestand van de bibliotheek en het gedrag van de API-methodes. Bovendien is het niet duidelijk of toekomstige API-methode-oproepen een effect hebben op de toestand van de collectie, hetgeen de cliënt zou kunnen waarnemen indien hij een verwijzing naar de collectie bijhoudt en de collectie inspecteert nadat hij nog verdere API-methode-oproepen gedaan heeft.

Analoog is het niet duidelijk wat het gedrag is van een API waarvan sommige methodes als resultaattype een wijzigbaar collectietype hebben. Hebben toekomstige API-methode-oproepen nog een effect op de teruggegeven collectie? Is het toegelaten dat de cliënt de teruggegeven collectie wijzigt, en zo ja, hebben dergelijke wijzigingen een effect op het gedrag van toekomstige API-methode-oproepen?

In beide gevallen (wijzigbaar collectietype als parametertype of resultaattype) is er mogelijk sprake van een wijzigbaar object naar hetwelk zowel de bibliotheek als de klant een referentie hebben.

In dergelijke gevallen is het belangrijk dat het gedrag van de bibliotheek duidelijk gespecificeerd wordt in de documentatie van de betreffende API-methodes. In het algemeen is dit moeilijk en vereist dit het gebruik van een geavanceerde logische taal zoals de *scheidingslogica* (Eng.: *separation logic*). Echter, in de meeste gevallen is er sprake ofwel van *permanente overdracht van eigendom*, ofwel van *tijdelijke overdracht van eigendom*, ofwel van een *live view*.

Permanente overdracht van eigendom in het geval van een parameter betekent dat de cliënt eigendom van de collectie permanent overdraagt aan de bibliotheek. Dit houdt in dat de cliënt zich ertoe verbindt de collectie na de API-methode-oproep nooit meer te inspecteren of te muteren.

Permanente overdracht van eigendom in het geval van een resultaat is analoog: het betekent dat de bibliotheek eigendom van de collectie overdraagt aan de cliënt, en de collectie dus niet meer zal inspecteren of muteren.

Permanente overdracht van eigendom geldt in het bijzonder indien de bibliotheek een kopie teruggeeft van een interne collectie.

Tijdelijke overdracht van eigendom is hoofdzakelijk van toepassing in het geval van een parameter: het betekent dat de cliënt eigendom van de collectie

overdraagt gedurende de looptijd van de API-methode-oproep, en bij het terugkeren van de oproep de eigendom terugkrijgt. Dit betekent dat de bibliotheek de collectie mag inspecteren en muteren gedurende de oproep, en enkel dan; de bibliotheek mag na afloop van de oproep de collectie niet meer inspecteren of muteren. Dit geldt in het bijzonder indien de bibliotheek een kopie neemt van een collectie die aangeleverd werd door de cliënt.

Een derde vaak voorkomende mogelijkheid is dat een collectie-object dat een API-methode als resultaat teruggeeft een *live view* is op de toestand van de bibliotheek. In dit geval is de toestand van de collectie gekoppeld aan de toestand van de bibliotheek: API-methodes die de toestand van de bibliotheek muteren hebben een overeenkomstig effect op de toestand van de collectie. Er zijn twee soorten live views: lees-schrijf-views en enkel-lezen-views. In het geval van lees-schrijf-views is het de cliënt toegelaten de mutator-methodes van de collectie op te roepen; deze operaties muteren dan niet enkel de toestand van de collectie maar ook die van de bibliotheek. In het geval van enkel-lezen-views is het de cliënt niet toegelaten de mutator-methodes van de collectie op te roepen; de toestand van de bibliotheek mag enkel gewijzigd worden door middel van de daartoe aangeboden API-methodes. Enkel-lezen-views zijn over het algemeen te verkiezen boven lees-schrijf-views, omdat hun gedrag eenvoudiger en duidelijker is.

Verfijnde collectietypes De situatie is veel eenvoudiger indien de collectietypes die voorkomen als parameter- of resultaattypes in een API onwijzigbaar zijn: aangezien noch de cliënt, noch de bibliotheek de collectie kan wijzigen, stelt de vraag van het al dan niet toegelaten zijn van mutaties, en het effect van mutaties, zich eenvoudigweg niet. Dikwijls is in dit geval het gedrag van een API-methode onmiddellijk duidelijk uit de methodenaam, de parameternamen, de parametertypes, en het resultaattype.

Het belangrijkste nadeel van het gebruik van onwijzigbare collectietypes in APIs is dat dit meestal een zekere performantie-kost heeft. In gevallen waar het belangrijk is dat er efficiënt omgegaan wordt met zeer grote collecties, of waar performantie kritisch is, is het gebruik van onwijzigbare collectietypes dan ook af te raden. Dikwijls betreft het echter kleine collecties, of betreft het niet-performantiekritische code; in die gevallen raden wij het gebruik van onwijzigbare collectietypes in APIs aan, omdat dit de betekenis van de API duidelijker en eenvoudiger maakt, en omdat dit het risico op bepaalde vaak voorkomende implementatiefouten elimineert. Het gebruik van onwijzigbare collectietypes kan de performantie soms zelfs ten goede komen, namelijk in de context van defensief programmeren: bij het gebruik van onwijzigbare collectietypes is het immers niet nodig preventief kopies te maken van binnenkomende of uitgaande collecties.

Voorbeelden van onwijzigbare collectietypes zijn het type ImmutableList en verwanten van Guava, en de types PList en PMap van purecollections. Een verschil tussen de Guava-types en de purecollections-types is dat deze laatste ook bedoeld zijn voor inwendig gebruik in de implementatie van een bibliotheek, om wijzigende verzamelingen van elementen bij te houden; hiertoe hebben de purecollections-types plus-, minus- en with-methodes ter vervanging van de add-, remove- en set-methodes van de overeenkomstige wijzigbare types.

Indien men er toch voor opteert een wijzigbaar collectietype te gebruiken,

zijn de types UnmodifiableList en verwanten van Apache Commons het overwegen waard. Qua functionaliteit is het type UnmodifiableList identiek aan de methode Collections.unmodifiableList; het voordeel van het gebruik van het type in de API-definitie is echter dat dit het zelf-documenterend gehalte van de API verhoogt, en daarnaast de programmeerfout elimineert waarbij per ongeluk een interne collectie onbeschermd doorgegeven wordt. Het blijft bij het gebruik van dit type echter noodzakelijk het gedrag van de API-methode zorgvuldig te documenteren, aangezien dit type het gedrag van de collectie niet volledig vastlegt.

Encapsulatie van collecties Het is één zaak de API van een bibliotheek goed te definiëren; het is nog een andere zaak de bibliotheek zo uit te werken dat ze de API correct implementeert. Het is een vaak voorkomende implementatiefout dat een interne collectie van een bibliotheek per ongeluk onbeschermd gelekt wordt naar de cliënt als resultaatwaarde van een inspectormethode, terwijl voor het gedrag van de inspectormethode een permanente overdracht van eigendom bedoeld wordt, waarbij het dus niet de bedoeling is dat wijzigingen die de cliënt aanbrengt in de teruggegeven collectie een effect hebben op de toestand van de bibliotheek. Dergelijke fouten kunnen vermeden worden door onwijzigbare collectietypes te gebruiken in de API. Als toch wijzigbare collectietypes gebruikt worden, moet men een kopie nemen van binnenkomende en uitgaande collecties, zodat er nooit sprake is van wijzigbare collecties naar dewelke zowel de cliënt als de bibliotheek een referentie hebben. Voor uitgaande collecties bestaat ook de alternatieve aanpak waarbij men de collectie inpakt in een enkel-lezen-verpakking (doorgaans een *unmodifiable wrapper* genoemd); dit dwingt men liefst af in de API door het type UnmodifiableList van Apache Commons of verwante types te gebruiken.

Referenties

[1] `http://code.google.com/p/capsules`.