KU Leuven
Departement Computerwetenschappen

Ontwerp van Softwaresystemen

Verslag iteratie 2

Team:
2

Mattias Buelens
Vital D'haveloose
Matthias Moulin
Ruben Pieters
Begeleider:
Mario Henrique Cruz Torres

Academiejaar 2013 – 2014

Samenvatting

Het project van het vak Ontwerp van Softwaresystemen uit de eerste master Computerwetenschappen (2013-2014) bestaat erin een gegeven softwaresysteem, namelijk het JUnit Framework (versie 4.11) te analyseren, te beoordelen, te refactoren en uit te breiden.

Dit verslag voor de tweede iteratiestap spitst zich toe op het ontwerpen, implementeren en testen van een uitbereiding voor dit systeem. Deze uitbereiding bestaat uit het automatisch uitvoeren van testen zodra nieuwe wijzigingen in de programmabroncode (die getest wordt) worden opgeslagen op schijf. Op die manier kunnen bugs sneller worden opgemerkt en hersteld. Daarnaast wordt er informatie bijgehouden over de uitvoer van de testen en de wijzigingen in de programmacode. Aan de hand van deze informatie kunnen de toekomstige testen in een bepaalde volgorde worden uitgevoerd.

Om deze functionaliteit te realiseren hebben we een pipeline geïmplementeerd. Deze roept andere objecten aan om veranderde klassen opnieuw in te laden, de veranderingen te detecteren en te verwerken, de testen te sorteren en uiteindelijk uit te voeren. De pipeline word op zijn beurt opgestart wanneer er een bepaalde tijd lang geen wijzigingen meer zijn gebeurd. Om informatie bij te houden over verschillende pipeline-runs heen werd een testdatabase geïmplementeerd. Daarnaast werd een systeem voorzien dat zowel Java-broncode als gecompileerde klassen (op schijf of in het geheugen) ter beschikking stelt aan de pipeline en zijn hulpobjecten.

De resulterende code werd ook getest. Met unit tests behaalden we een coverage van 73%. Er werd daarbij gefocust op de normale werking.

Inhoudsopgave

Inleiding

Het volledige proces vanaf het detecteren van wijzigingen in de broncode tot het uitvoeren van de bijhorende testen en het opslaan van de testresultaten wordt uitgevoerd door één daemon. Dit proces is opgedeeld in verschillende sequentiële stappen en heeft dus een pipeline structuur. De verschillende stappen zijn:

1. De pipeline wordt gevoed met `StoreEvent`s. Dit zijn toevoegingen, verwijderingen of aanpassingen van de elementen in een abstracte `Store`. Deze `Store` is een soort naming registry voor klassen (broncode of binaries) die het werken met een opslagmedium (bestandssysteem of geheugen) afschermt voor de verdere implementatie. De `Store` wordt in detail besproken in sectie 1.1. De veranderingen worden opgemerkt met een `StoreWatcher` zoals uitgelegd in sectie 1.3.

2. De gebruikte `ClassLoader` wordt herladen. Dit is noodzakelijk om de aangepaste (recentste) versie van de code te kunnen inladen. De werking hiervan wordt uitgelegd in sectie 1.5.

3. Het identificeren en behandelen van veranderingen in de broncode en testcode. Dit wordt mogelijk gemaakt door de code te compileren. De oude en nieuwe code wordt vergeleken en met de resultaten van de vergelijkingen wordt de `TestDatabase` geüpdatet. De database zelf wordt uitgelegd in sectie 1.2, de detectie van veranderingen komt aan bod in sectie 1.6.

4. Het sorteren van de testen volgens een bepaald criterium wordt besproken in sectie 1.7.

5. Het uitvoeren van de testen met behulp van het JUnit framework wordt uitgelegd in sectie 1.8.

Het delegeren van deze verschillende stappen door de instantie van `Pipeline` wordt weergeven in het UML-sequentiediagram van figuur 1. Hoe andere objecten de gedelegeerde taken vervolgens uitvoeren wordt uitgebreid toegelicht in sectie 1.

De pipeline zal niet direct na een wijziging in de broncode een nieuwe pipeline-run uitvoeren. Dit zou voor een overdaad aan (zinloze) uitvoeringen kunnen zorgen wanneer veel bestanden tegelijk veranderen, bijvoorbeeld bij een refactoring. In de plaats daarvan wordt bij elke binnenkomende verandering een toekomstige pipeline-run aangevraagd of uitgesteld, zodat een pipeline-run enkel kan starten als er voor een bepaalde tijd geen nieuwe wijzigingen meer zijn. De `DeferredTaskRunner` realiseert dit gedrag, zoals uitgelegd in sectie 1.4.

Over verschillende pipeline-runs heen moet informatie worden bijgehouden. Dit is de verantwoordelijkheid van de `TestDatabase`. Een instantie van deze klasse houd tests (`Test`), geteste methodes (`TestedMethod`) en de links tussen de twee bij. Daarnaast wordt informatie die relevant is voor de policies bijgehouden in de passende klassen. Hoe dit precies gebeurt wordt verder uitgelegd in sectie 1.2.

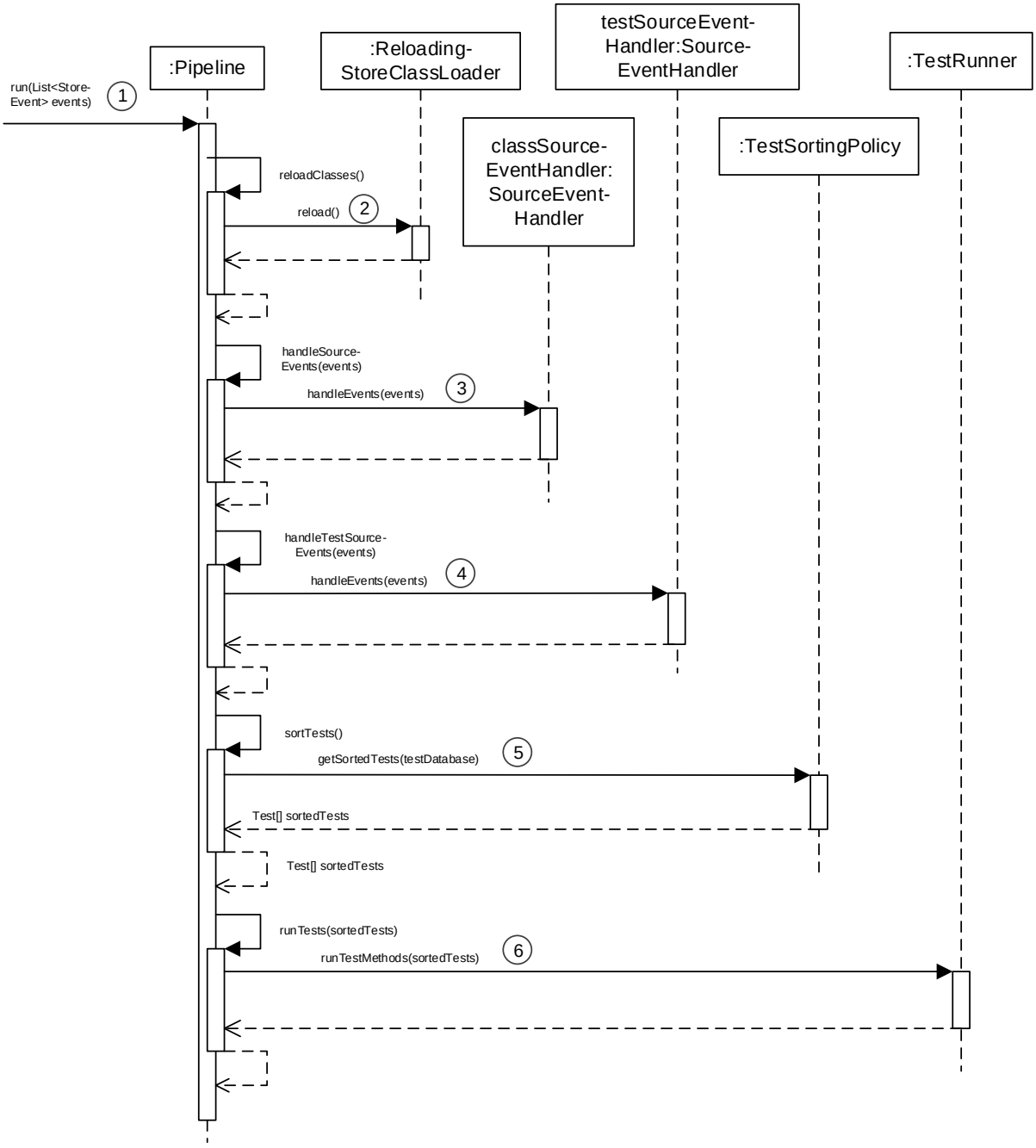
1 Ontwerp

In deze sectie gaan we dieper in op hoe de onderdelen van de pipeline precies in elkaar zitten en waarom we ze zo ontworpen hebben. De eerste twee secties behandelen respectievelijk het gebruik van `Store`s en de `TestDatabase`. De verschillende stappen van de `Pipeline` worden beschreven in de daarop volgende subsecties.

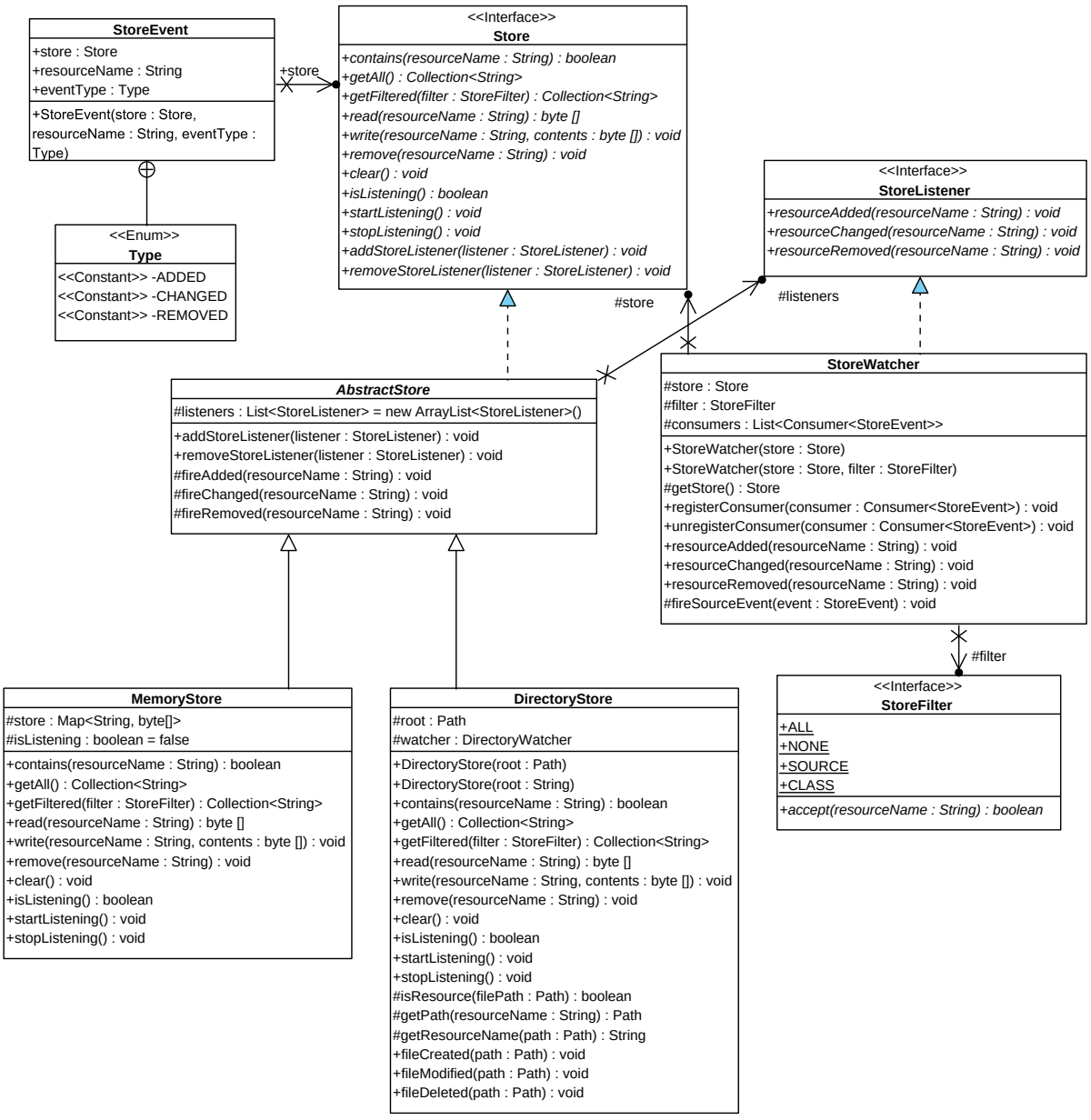
1.1 Programmacode en binaries beschikbaar maken via `Stores`

De `Store` houdt een mapping tussen resourcenamen en resources bij. Het UML-klassendiagram van de `Store` hiërarchie is weergegeven in figuur 2.

De `Store` interface definiëert methodes eigen aan een algemene `Store` instantie. Een concrete `Store` bevat een collectie van resources waaruit bytes gelezen kunnen worden (read) en waar naar toe geschreven kan worden (write). Resources kunnen uit de `Store` verwijderd worden (remove). We kunnen opvragen of een `Store` een bepaalde resource bevat (contains). We kunnen de naam van alle resources in de store opvragen (getAll). Daarnaast kunnen `StoreListeners` toegevoegd worden aan en verwijderd worden van een `Store`. Deze `StoreListeners` kunnen reageren op het verwijderen uit, toevoegen aan of wijzigen van resources van een bepaalde `Store`.



Figuur 1: UML-Sequentiediagram van het delegeren in de pipeline. De externe methode-aanroepen worden geëncapsuleerd in eigen methodes. Korte uitleg over deze methode-aanroepen:
1. `run(List<StoreEvent> events)`: De lijst van `StoreEvent`s bevat de gebeurtenissen in de `Store`. Deze worden meegegeven aan de `Pipeline` om tijdens de run correct afgehandeld te worden.
2. `reloadClasses()`: De `ClassLoader` wordt herladen. Dit is noodzakelijk om de aangepaste (recentste) versie van de code te kunnen inladen.
3. `handleSourceEvents(events)`: Deze methode zal (indirect) alle `.java` resources (waarnaar verwezen wordt in de events) compileren en deze in de juiste `Store` plaatsen. Vervolgens zal de `TestDatabase` geüpdatet worden.
4. `handleTestSourceEvents(events)`: Alle veranderde resources zullen doorzocht worden op testmethodes. Die methodes worden als `Test`s aan de `TestDatabase` toegevoegd.
5. `getSortedTests(testDatabase)`: De testen worden gesorteerd aan de hand van de gekozen `TestSortingPolicy`.
6. `runTestMethods(sortedTests)`: De testen worden uitgevoerd met behulp van het JUnit Framework.



Figuur 2: Een UML-klassendiagram van het `Store` deelsysteem.

Twee concrete implementaties van de `Store` komen voor in ons project: de `DirectoryStore` en de `MemoryStore`. Deze eerste `Store` maakt een abstractie van een bestandssysteem. De basis read/write/remove/contains operaties worden omgezet naar de juiste bestandssysteemaanroepen. Om haar `StoreListeners` op de hoogte te kunnen brengen wordt gebruik gemaakt van de `DirectoryWatcher` klasse. Deze `DirectoryWatcher` klasse maakt gebruik van de `java.nio.file.WatchService` en de `java.nio.file.WatchKey` om de volgende gebeurtenis in het bestandssysteem te kunnen opvragen. De `DirectoryWatcher` brengt de `DirectoryStore` op de hoogte die op haar beurt haar `StoreListeners` inlicht over de gebeurtenissen in het bestandensysteem.

De `MemoryStore` maakt een abstractie van het geheugensysteem. De basis read/write/remove/contains operaties worden omgezet naar operaties op een `Map` collectie van de `MemoryStore`. De `MemoryStore` licht zijn `StoreListeners` in zonder eerst zelf ingelicht te worden. Dit is in contrast met de `DirectoryStore` omdat de `MemoryStore` de enige is die zijn resources (geheugenplaatsen) kan aanpassen, moet deze niet door derden ingelicht worden.

De `Store` hiërarchie wordt vergezeld van enkele hulpklassen:

- Een `StoreEvent` representeert een bepaalde gebeurtenis in een `Store`. Dit event bevat de `Store` waarin het gebeurt is, de resourcenaam waarop het van toepassing is en een type van het event in kwestie. De mogelijke types zijn: toevoeging, wijziging of verwijdering.
- De `StoreWatcher` zal deze `StoreEvents` doorgeven aan zijn `Consumer`s. De `StoreWatcher` is een `StoreListener` die in zijn listenerimplementaties een `StoreEvent` creëert en doorgeeft aan zijn geregistreerde `Consumer`s.
- Een `Consumer` is een klasse die data van een bepaald type zal consumeren via haar `consume(T item)` methode.

1.2 Informatie over testen en methodes bewaren in de `TestDatabase`

Om testen te kunnen ordenen volgens bepaalde criteria, moet bepaalde informatie over deze testen bijgehouden worden. Dit is de verantwoordelijkheid van de klasse `TestDatabase`. Voor de geïmplementeerde policies moet het volgende opgevraagd kunnen worden:

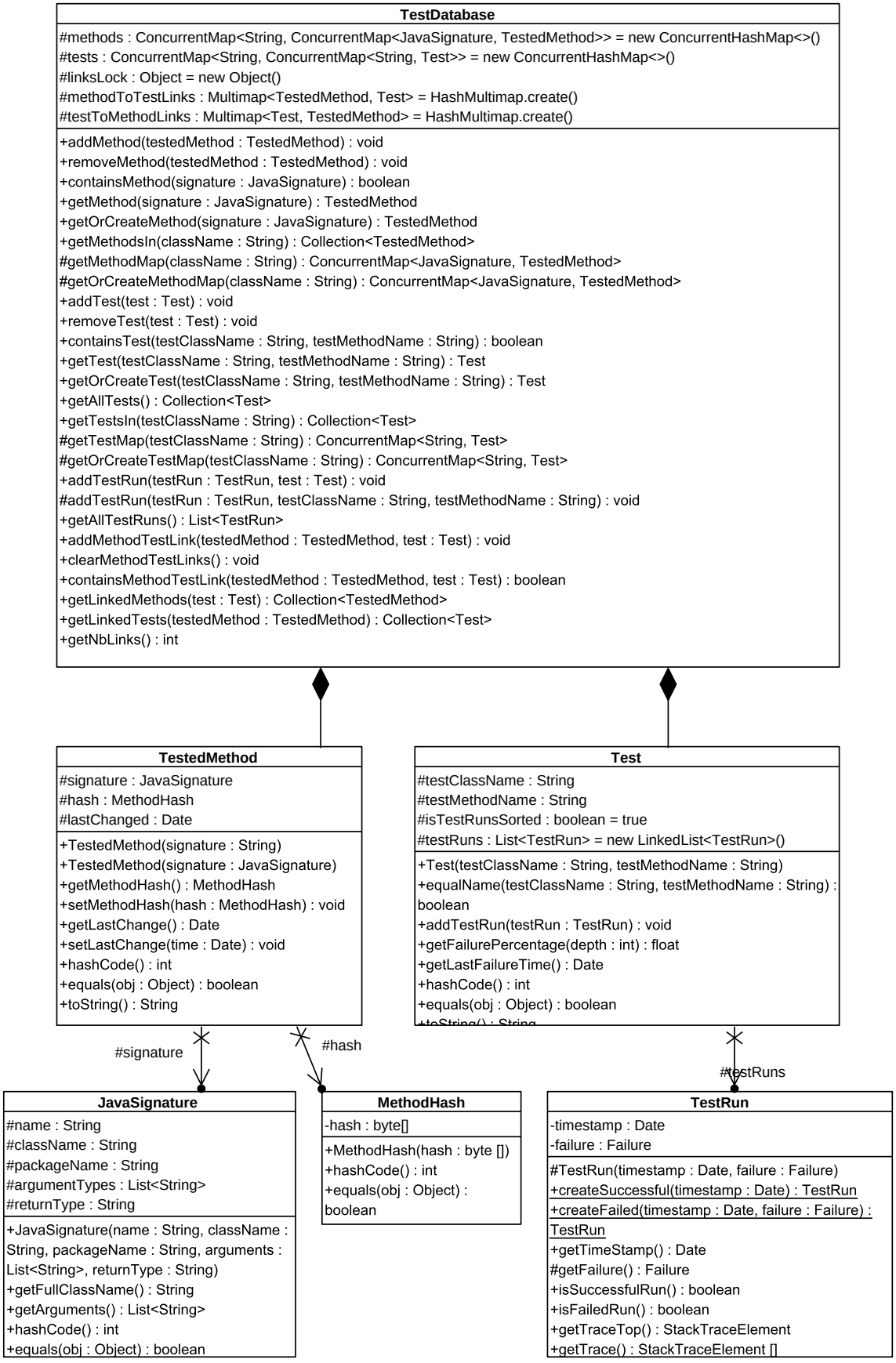
- **Changed Code First**: van elke methode die getest wordt de tijd van haar laatste wijziging.
- **Distinct Failure First**: van elke gefaalde `Test` de stack trace om falingen te kunnen groeperen.
- **Last Failure First**: van alle `Test`s de tijd van hun laatste faling.
- **Frequent Failure First**: van alle `Test`s het falingspercentage van de laatste n `TestRun`s.

Om deze en mogelijks nog andere toekomstige policies van de nodige informatie te voorzien wordt een structuur gebruikt zoals weergeven in het UML-klassendiagram in figuur 3. Een `TestDatabase` instantie beschikt over verschillende collecties voor `Test`s en `TestedMethod`s. `Map`s laten toe om snel `TestedMethode`s en `Test`s te vinden aan de hand van hun klassennaam en signatuur respectievelijk klassennaam en methodenaam. Twee `com.google.common.collect.MultiMap`s tussen de `TestedMethode`s en `Test`s en omgekeerd verzorgen de many-to-many relatie tussen testen en methoden en laten snelle toegang vanuit beide richtingen toe. Toegang tot deze datastructuren wordt verzorgd in de publieke methodes van `TestDatabase`.

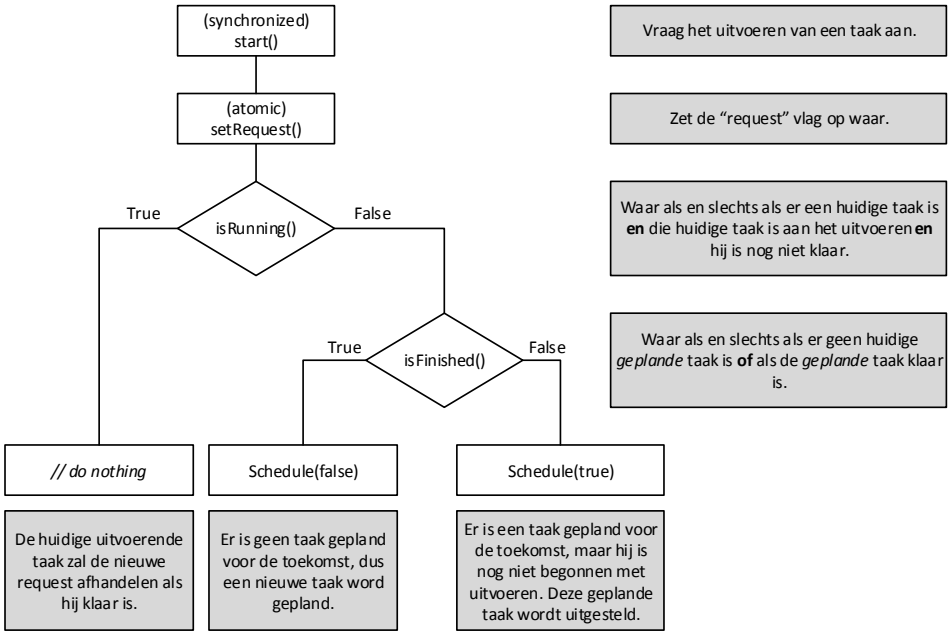
Informatie over de individueel geteste methodes en testmethodes wordt door objecten van de klassen `TestedMethod` respectievelijk `Test` bijgehouden. Voor de vier policies hierboven wordt de nodige informatie op vrij triviale wijze bijgehouden. Een geteste methode beschikt over de tijd van zijn laatste wijziging. Een testmethode kan aan de hand van een lijst van `TestRun`s het tijdstip van zijn laatste faling achterhalen, alsook het falingspercentage van de laatste n runs.

1.3 Veranderingen in de (test)code opmerken

De `DirectoryWatcher` klasse moet het bestandssysteem in de gaten houden en meldingen aan zijn listeners geven wanneer veranderingen zich voordoen. Om dit te realiseren gebruikt de `DirectoryWatcher`



Figuur 3: Een UML-klassendiagram van het testdatabase deelsysteem.



Figuur 4: Flow-chart van de `Start` methode van de `DeferredTaskRunner`.

de `java.nio.file.WatchService` klasse. Deze heeft een `take` methode die wacht op een volgende `java.nio.file.WatchKey`. Deze `WatchKey` bevat allerlei informatie over de gebeurtenis die is opgetreden. Aan de hand van die informatie kunnen we de `DirectoryWatchListener`s inlichten over die gebeurtenis. De `DirectoryStore` implementeert de `DirectoryWatchListener` interface.

De `DirectoryWatcher` wordt gebruikt in de `DirectoryStore` om indirect de functionaliteit in te vullen van het op de hoogte brengen van de `StoreListener`s. Dit wordt verwezenlijkt door de `DirectoryStore` zelf te laten watchen op het bestandensysteem via de `DirectoryWatcher`. Wanneer een gebeurtenis plaatsvindt, worden haar `StoreListener`s op de hoogte gebracht nadat ze zelf op de hoogte gebracht is door haar `DirectoryWatcher`. De `MemoryStore` zal bij haar read en write operaties zelf bepalen wanneer zij haar `StoreListener`s moet inlichten aangezien we er van uitgaan dat de `MemoryStore` zelf de enige is die over haar geheugenruimte beschikt en er dus veranderingen in kan aanbrengen.

Uiteindelijk worden de `StoreWatcher`s gebruikt om veranderingen in de programma-broncode te detecteren die opgebouwd zijn uit de hierboven uitgelegde structuur. Deze `StoreWatcher`s krijgen een `DeferredConsumer` mee die de `StoreEvents` (die de `StoreWatcher`s gecreëerd heeft op basis van wat ze hoort) in batches zal 'consummeren'.

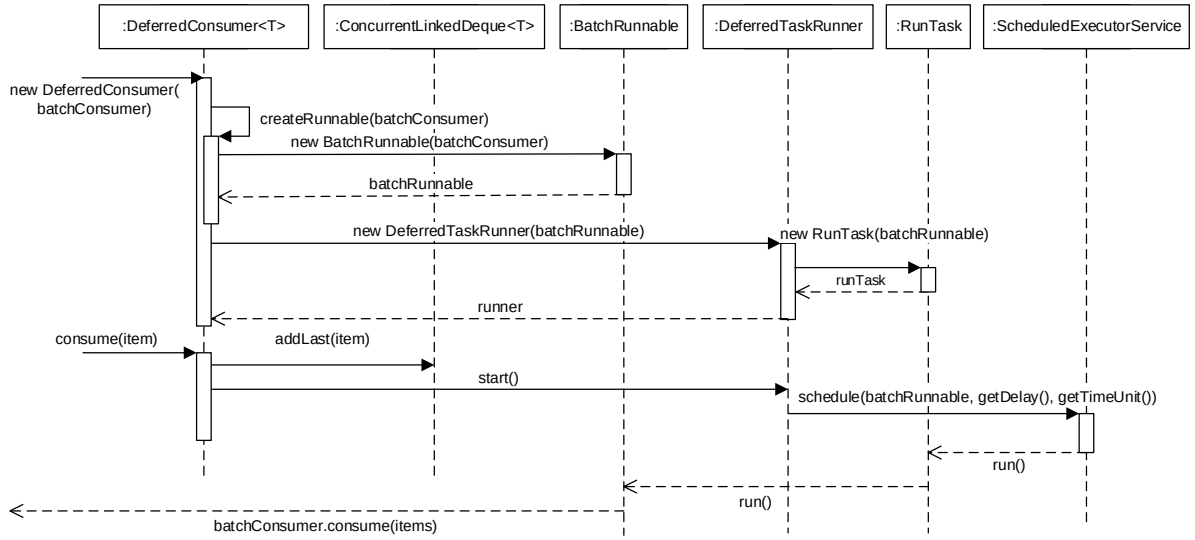
1.4 De pipeline uitgesteld starten

De pipeline zal niet onmiddellijk na een wijziging in de broncode een nieuwe pipeline-run uitvoeren. In de plaats daarvan wordt er een toekomstige pipeline-run aangevraagd en gepland indien er een pipeline-run bezig is of indien er niets gepland is. Als er reeds een pipeline-run gepland is (en deze is niet bezig), wordt deze herpland naar een later tijdstip. Als er in een bepaald tijdsinterval geen aanvragen meer binnenkomen, zal de pipeline-run uiteindelijk uitgevoerd worden. De `DeferredTaskRunner` realiseert dit gedrag.

We willen tegengaan dat elke wijziging uit een bulk van sourcewijzigingen een aparte pipeline-run zou kunnen starten. Bulk van sourcewijzigingen komen voor wanneer bijvoorbeeld een refactoring meerdere wijzigingen veroorzaakt of wanneer frequent gesaved wordt.

De klasse `DeferredTaskRunner` zal een taak op deze uitgestelde manier uitvoeren. De `start` methode vraagt een request voor een run aan. Vanaf hier zijn er drie mogelijkheden dewelke weergegeven zijn op de flow-chart van figuur 4 weergegeven.

De `DeferredTaskRunner` is een algemene implementatie om taken uitgesteld te kunnen plannen. Daarnaast is er ook nood om data te verzamelen en om deze vervolgens te kunnen consummeren. De



Figuur 5: UML-sequentiediagram voor de creatie van een `DeferredConsumer` alsook de aanroep van de `consume` methode. Merk op dat niet elke stap is weergegeven vanaf het aanroepen van de `start` methode van de `DeferredTaskRunner`. De `DeferredTaskRunner` maakt gebruik van een threadpool waaraan een `RunTask` wordt toegevoegd. Deze `RunTask` kapselt de `BatchRunnable` in en bevat nog wat velden nodig voor het beheren van de planning.

`DeferredConsumer` maakt gebruik van de `DeferredTaskRunner`, buffert de data om deze bij het uitvoeren van de taak door te geven aan zijn (batch) `Consumer` die deze lijst van data kan behandelen/afhandelen. Deze functionaliteit is geïmplementeerd in de `DeferredConsumer`. Deze is zelf een `Consumer<T>` en heeft listeners van het type `Consumer<List<T>>`. De `List` is noodzakelijk omdat de volgorde van de binnenkomende data belangrijk is. De `DeferredConsumer` maakt gebruik van de `DeferredTaskRunner` in haar eigen `consume` methode. In deze methode zal de lijst van huidige items geüpdatet worden en een run gepland worden. Deze run houdt in dat alle listeners van de `DeferredConsumer` zullen ingelicht worden met alle items in de huidige itemlijst. Dit is weergeven in figuur 1.

De data wordt in ons project voorgesteld door `StoreEvent`s en de `Pipeline` levert de eigenlijke implementatie voor de `Consumer<List<StoreEvent>>`. In ons ontwerp laten we slechts één zo'n consumer toe op een `DeferredConsumer`, maar de uitbreiding naar meerdere consumers is haalbaar indien dit later nodig blijkt (bijvoorbeeld voor verschillende parallelle pipelines).

1.5 Klassen (her)inladen

Om de klassen die gecompileerd zijn te testen moeten ze ingeladen worden in de huidige virtuele machine. Aangezien de gecompileerde klassen in de `Store`s zitten is er nood aan een `StoreClassLoader`. Deze `StoreClassLoader` gaat simpelweg de gecompileerde binaries opzoeken via de resourcenaam en deze dan inladen via de `defineClass` methode van de `java.lang.ClassLoader`.

Aangezien verschillende versies van klassen moeten ingeladen worden is er nood aan het herladen van die klassen. Dit moet gebeuren door de `ClassLoader` in te kapselen en wanneer we een klasse willen herladen een nieuwe ClassLoader inzetten. Dit is nodig omdat de `ClassLoader` geen ondersteuning biedt om een reeds ingeladen klasse te vervangen. Deze functionaliteit is geïmplementeerd in de `ReloadingClassLoader`. De `ReloadingStoreClassLoader` erft dan over om deze functionaliteit aan te bieden voor `StoreClassLoader`s.

1.6 Veranderingen in de code verwerken

Het behandelen van events wordt algemeen gedefiniëerd door de abstracte klasse `SourceEventHandler`. Deze specifieert een algemeen gedrag: een `handleEvents(List<StoreEvent>)` methode. Deze wordt in de volgende secties voor de subklassen in detail uitgelegd.

Daarnaast wordt de `collectChanges` methode aangeboden. Omdat de test daemon uitgesteld wordt uitgevoerd, kunnen meerdere `StoreEvent`s over één resource in de lijst voorkomen. De `collectChanges` methode doorloopt alle gegeven `StoreEvent`s in volgorde en maakt een `Changes` object die lijsten van alle *uiteindelijk* toegevoegde, aangepaste en verwijderde resources bevat. Wanneer bijvoorbeeld een resource eerst gewijzigd en dan verwijderd is, zal deze uiteindelijk enkel in de lijst van verwijderde resources voorkomen.

1.6.1 Veranderingen in de broncode verwerken

De `ClassSourceEventHandler` implementeert `handleEvents` voor events in de broncode. Dit proces gebeurt in verschillende stappen:

1. Alle klassen horende bij verwijderde bronbestanden worden uit de `TestDatabase` verwijderd. Dit gebeurt met behulp van de `MethodChangeUpdater`.

2. Alle toegevoegde of aangepaste bronnen worden opnieuw gecompileerd. Hiervoor maken we gebruik van de Java compiler uit de Eclipse Java Developer Tools[3].

3. Toegevoegde, aangepaste of verwijderde methoden in de klassen worden met de `MethodChangeUpdater` gedetecteerd en doorgevoerd in de `TestDatabase`.

Om de aangepaste code te detecteren maken we gebruik van een hash van de gecompileerde methodes. Zo kunnen we veranderingen die effect hebben op de gecompileerde klasse opvangen en zullen we onnodig werk vermijden als er enkel visuele veranderingen gemaakt worden (indentatie, documentatie. . .).

De `MethodHasher` zal met behulp van een `ClassReader` uit de ASM library [2] elke methode van een gecompileerde klasse overlopen om van de methodebytes dan een hashwaarde terug te geven.

De `MethodChangeUpdater` kan dan de oude hashwaarden uit de `TestDatabase` opvragen en vergelijken met de nieuwe hashwaarden. Als deze verschillen, wordt de bijhorende `TestedMethod` in de database bijgewerkt met de nieuwe hash waarde en een nieuwe tijd van laatste wijziging ingesteld.

1.6.2 Veranderingen in de testcode verwerken

De `TestSourceEventHandler` implementeert `handleEvents` voor events in de testcode. Dit proces is analoog aan dat van de `ClassSourceEventHandler`, maar gebruikt nu een `TestChangeUpdater` in plaats van een `MethodChangeUpdater`. Deze moet geen hashes berekenen om de tijd van de laatste wijziging te bepalen, maar moet wel enkel eigenlijke testmethoden toevoegen aan de database (en geen hulpmethoden met bijvoorbeeld `@Before`).

Om de aangepaste tests te detecteren, gebruikt de `TestChangeUpdater` een JUnit `Runner` om de testmethoden uit een klasse te halen. Aan de hand van de `Description` van de aangemaakte runner kunnen we nagaan welke methoden testen zijn. De nieuwe lijst van testmethoden in de klasse worden dan vergeleken met de oude lijst uit de database, waarna toevoegingen en verwijderingen gevonden en doorgevoerd kunnen worden naar de database. We controleren dus niet zelf op de aanwezigheid van `@Test` annotaties (JUnit 4) of methodenamen beginnend met `test` (JUnit 3), maar gebruiken de bestaande JUnit `RunnerBuilder` architectuur. We behouden zo dezelfde compatibiliteit met oudere testklassen als JUnit zelf.

1.7 Testen sorteren

`Test` objecten kunnen in de huidige versie volgens vier verschillende criteria gesorteerd worden. Dit wordt verwezenlijkt door de volgende `TestSortingPolicy` implementaties:

- `ChangedCodeFirst`: De `Test`s die gewijzigde code uitvoeren worden eerst uitgevoerd. De `Test`s worden gesorteerd volgens de meeste recentste wijzigingsdatum van al hun corresponderende gelinkte `TestMethod`s. De koppeling tussen `Test`s en `TestMethod`s wordt bekomen via de `TestDatabase`.

- `DistinctFailureFirst`: Vaak gebeurt het dat een fout in de code meerdere `Test`s doet falen. Dit kan gedetecteerd worden door de stack trace van de falende `Test`s te vergelijken. Op die manier kunnen deze opgedeeld worden in groepen die waarschijnlijk dezelfde oorzaak hebben. Met deze

`TestSortingPolicy` moet men eerst van elke groep een test uitvoeren voor men aan de rest begint. Het eerste `StackTraceElement` van elke gefaalde `TestRun` per `Test` wordt opgevraagd. Is dit de eerste maal dat zo'n `StackTraceElement` voorkomt en komt de `Test` in de huidige ordening niet voor dan wordt deze toegevoegd aan de ordening. `Test`s die geen enkel nieuw `StackTraceElement`s opleveren (omdat ze altijd slaagden) worden elders bijgehouden en zullen uiteindelijk als laatste toegevoegd worden aan de ordening. De `TestRun`s worden opgevraagd aan de `Test` zelf. Verder is er een diepte-parameter voorzien om slechts de laatste n `TestRun`s van elke `Test` in rekening te nemen, opdat oude falingen geen blijvende rol meer zullen spelen.

- `FrequentFailureFirst`: De `Test`s die het vaakst faalden in hun laatste n `TestRun`s worden eerst uitgevoerd. Het falingspercentage van de laatste n `TestRun`s wordt opgevraagd aan de `Test` zelf.

- `LastFailureFirst`: De `Test`s die laatst faalden worden eerst uitgevoerd. De datum van de laatste faling wordt opgevraagd aan de `Test` zelf.

Het UML-klassendiagram van de vier implementerende `TestSortingPolicies` alsook de `TestSortingPolicy` interface zijn weergeven in figuur 6. De `FrequentFailureFirst` en `LastFailureFirst` policies moeten enkel met de `Test`s zelf werken zonder tussenkomst met de `TestDatabase` en zonder `TestRuns` van de `Test`s op te vragen. Daarom implementeren ze de `Comparator<Test>` interface.

Een `TestSortingPolicy` heeft de volgende rechten en plichten:

- Hij mag enkel de aangeboden query/getter-methoden van de gekregen `Test`s en `TestDatabase` gebruiken.

- Hij is niet bevoegd om de toestand van deze gekregen objecten te wijzigen.

- Hij mag de meegegeven `Test[]` of `Collection<Test>` niet wijzigen.

Deze plichten zijn in ons ontwerp impliciet. Indien vereist kunnen deze expliciet gemaakt worden door beperkte views op de gegeven data te leveren zonder mutatormethoden, zodat wijzigingen onmogelijk zijn.

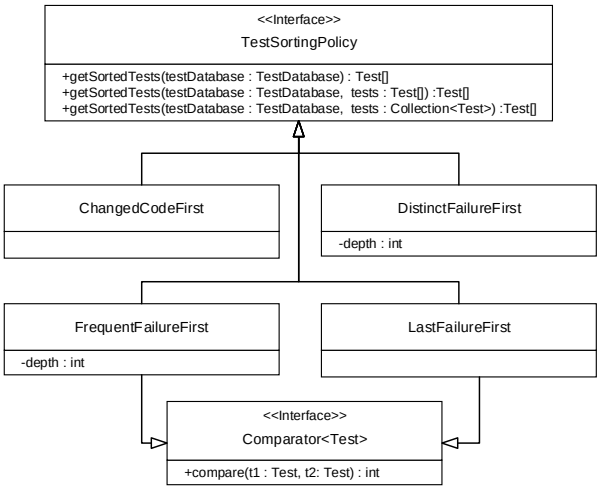
Vanaf JUnit versie 4.11 wordt er gebruik gemaakt van een deterministische (maar niet voorspelbare) testmethode volgorde bij het uitvoeren. Het is echter mogelijk een volgorde op te leggen in de testklasse door gebruik te maken van de `@FixMethodOrder` annotatie. Dit is de enige JUnit functionaliteit die we niet ondersteunen in ons ontwerp, omdat wij de volgorde van de testen laten afhangen van de gekozen policy. Anderzijds wordt het sterk afgeraden om een vaste volgorde voor unit tests op te leggen, omdat dit vaak wijst op slecht ontworpen testklassen met onderlinge afhankelijkheden tussen de testmethoden.

1.8 Gesorteerde testen uitvoeren

Eenmaal de volgorde voor het uitvoeren van de tests is vastgelegd kunnen de testen worden uitgevoerd. De `TestRunner` is verantwoordelijk voor het runnen van elke `Test` in de volgorde waarin ze hem gegeven worden. In het UML-sequentiediagram van figuur 7 is weergegeven hoe de `Pipeline` de `TestRunner` aanroept om dit te doen. De `TestRunner` vraagt een `Request` voor elke aparte `Test` op en zal vervolgens dit `Request` uitvoeren met zijn `JUnitCore`. In het UML-sequentiediagram is nu ook mooi de koppeling tussen onze daemon en het bestaande JUnit Framework weergegeven.

Omdat elke `Test` apart uitgevoerd wordt door de `TestRunner` in plaats van als één testrun, ontvangen de luisterende `RunListener`s op het `JUnitCore` object meerdere beginnen en eindes van testruns in één 'volledige' run. Dit neveneffect van de uitvoeringsstrategie van de `TestRunner` is niet conform met het verwacht contract voor de `RunListener`s, en dus moet de `TestRunner` ook de `RunListener`s anders gaan beheren.

De `TestRunner` maakt bij constructie een `TestRunListener` aan en geeft deze mee aan zijn `JUnitCore`. Deze `TestRunListener` stuurt zijn methodeaanroepen door naar de geregistreerde `RunListener`s van de `TestRunner`. De `JUnitCore` krijgt dus maar één `RunListener` mee, die op zijn beurt de `RunListener`s die opgeslagen worden op het niveau van de `TestRunner` zal verwittigen bij de gebeurtenissen *behalve* bij het begin en het einde van één enkele testrun. Het begin van de eerste `Test` uitvoering en het einde van de laatste `Test` uitvoering worden apart verwittigt door de `TestRunner` zelf. Op deze manier wordt het implementatiedetail dat testen apart uitgevoerd worden afgeschermd van de buitenwereld. Nadat



Figuur 6: UML-klassendiagram van de `TestSortingPolicy` hiërarchie. De `getSortedTests` methode die enkel een `TestDatabase` als argument verwacht wordt gebruikt om alle `Test`s van de `TestDatabase` te sorteren en dus uiteindelijk uit te voeren. De `getSortedTests` methodes die een `Array` of `Collection` met `Test`s verwachten worden gebruikt om slects de meegegeven subset van `Test`s te sorteren en dus uiteindelijk uit te voeren. Deze functionaliteit kan eenvoudig worden ingepast in het grotere prentje van figuur 1.

alle testen zijn afgelopen wordt een `Result` object aan de `RunListener`s (van de `TestRunner`) gegeven die alle resultaten bevat alsof alle testen in één geheel werden uitgevoerd, conform met het verwachte contract.

1.9 Patronen en principes

1.9.1 GoF patronen

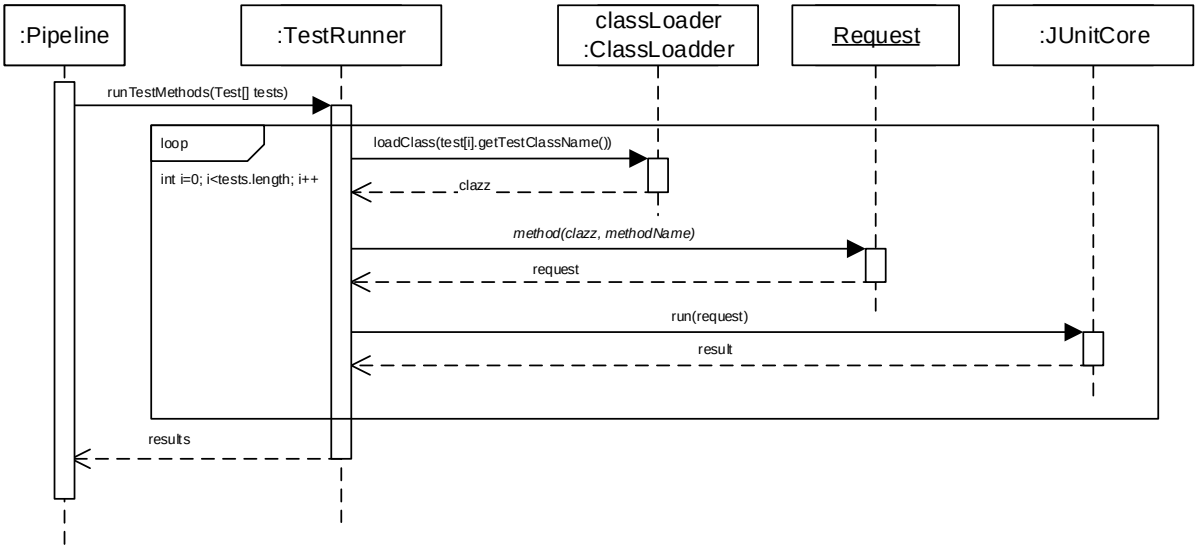
In ons project gebruikten we onder andere de volgende GoF patronen:

- **Decorator**: `ReloadingClassLoader`
  Het herladen van klassen in een `ClassLoader` kan moeilijk aan een bestaande `ClassLoader` toegevoegd worden. Een `ClassLoader` laat niet toe om reeds ingeladen klassen weer te verwijderen. Als oplossing hiervoor wordt een `ClassLoader` gedecoreerd die vervangen kan worden door een nieuwe instantie wanneer de klassen herladen moeten worden.

- **Factory Method**: `ReloadingClassLoader`
  Telkens de `ReloadingClassLoader` wordt herladen, heeft deze een nieuwe instantie nodig van een `ClassLoader`. Daarvoor biedt hij een factory methode aan waarin subklassen zo'n concrete `ClassLoader` moeten maken. De `ReloadingStoreClassLoader` implementeert dit bijvoorbeeld door een nieuwe `StoreClassLoader` te maken.

- **Observer**: `DirectoryWatchListener`, `StoreWatcher`, `TestDatabaseListener`
  Observers (of Listeners) komen courant voor in softwaresystemen, en ook in dit softwareproject bewijzen ze hun nut. Verschillende externe partijen kunnen geïnteresseerd zijn in gebeurtenissen op een object (zoals een directory, store of test database) maar deze partijen mogen niet gekend (en dus sterk gekoppeld) zijn door dat object. Het registreren van een `Listener` zorgt voor een lossere koppeling tussen het object en de externe partij(en).
  Het `TestRunsModel` registreert bijvoorbeeld een `TestDatabaseListener` bij de `TestDatabase` om zijn beeld op de `TestRun`s up-to-date te houden, zonder dat de database weet heeft van het bestaan van een `TestRunsModel`.

- **Adapter**: `StoreWatcher`, `DirectoryStore`
  De `Store` biedt een `StoreListener` aan om anderen op de hoogte te brengen van wijzigingen in



Figuur 7: UML-sequence diagram voor het uitvoeren van tests. Deze functionaliteit kan eenvoudig worden ingepast in het grotere diagram van figuur 1.

de resources. Echter, de pipeline wil deze wijzigingen pas *later* verwerken en kan dus niet onmiddellijk reageren als een `StoreListener`. De `StoreWatcher` slaat de brug hiertussen door de binnenkomende `StoreListener` aanroepen om te zetten tot consumeerbare `StoreEvent` objecten. Deze kunnen dan later geconsumeerd worden door de pipeline. De `StoreWatcher` zet dus de `add/removeListener` interface voor `StoreListener`s om naar een `register/unregisterConsumer` interface voor `Consumer<StoreEvent>`s.
De `DirectoryStore` communiceert met een `DirectoryWatcher`. Deze laatste gebruikt `DirectoryWatchListener`s, terwijl de `DirectoryStore` `StoreListener`s moet accepteren. De `DirectoryStore` fungeert dus als adapter tussen deze twee incompatibele listener interfaces.

- **Strategy**: `TestSortingPolicy`
  Testen kunnen op verschillende manieren gesorteerd worden, en de precieze `TestSortingPolicy` moet makkelijk veranderd kunnen worden. Ook moeten nieuwe policies toevoegen mogelijk zijn. Door policies als objecten te behandelen (in plaats van als methoden), kunnen objecten van deze klasse doorgegeven worden aan de pipeline om de policy te veranderen.

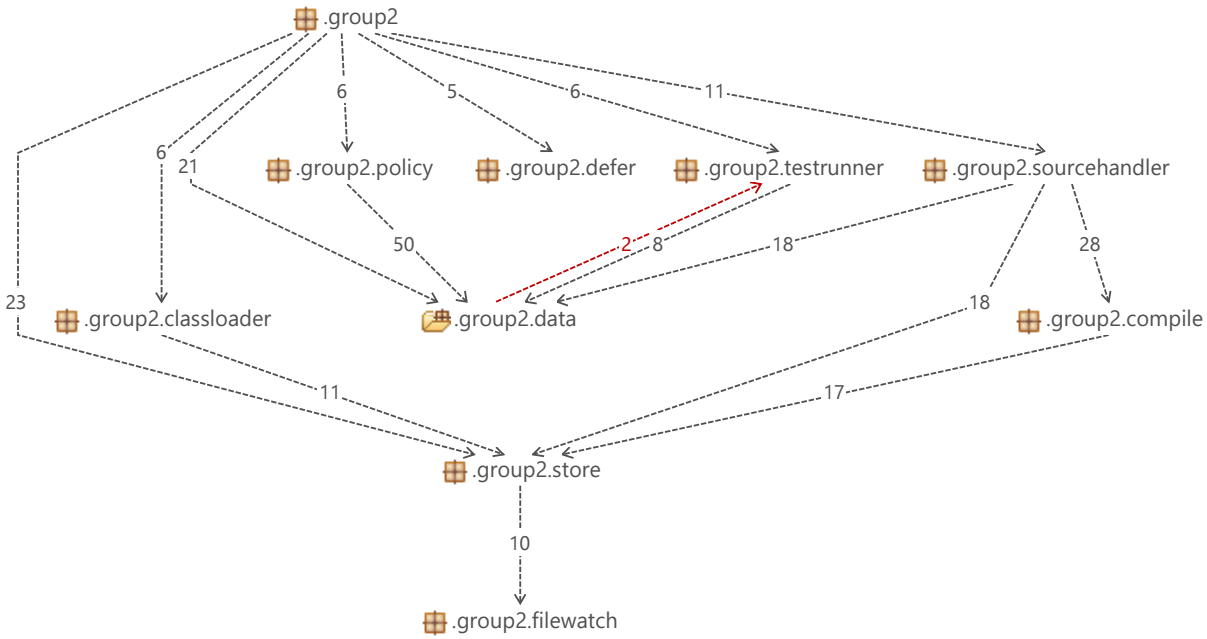
- **Singleton**: `OssRewriterLoader`
  Om technische redenen mag er slechts één `OSSRewriter` agent aan de huidige virtuele machine gekoppeld worden. Met een singleton wordt deze beperking afgedwongen: er kunnen geen andere `OssRewriterLoader`s aangemaakt worden buiten de instantie aangeboden met de statische `getInstance` methode. In onze implementatie wordt deze instantie "lui" aangemaakt, m.a.w. bij de eerste keer dat deze effectief opgevraagd wordt.

- **Facade**: `Pipeline`
  De `Pipeline` biedt een sterk vereenvoudigde interface om met het volledige systeem te communiceren. Het instantieert zelf de nodige objecten en zet de nodige relaties op tussen deze objecten. Gebruikers kunnen met eenvoudige `start` en `stop` aanroepen het hele systeem aansturen.

1.9.2 GRASP principes

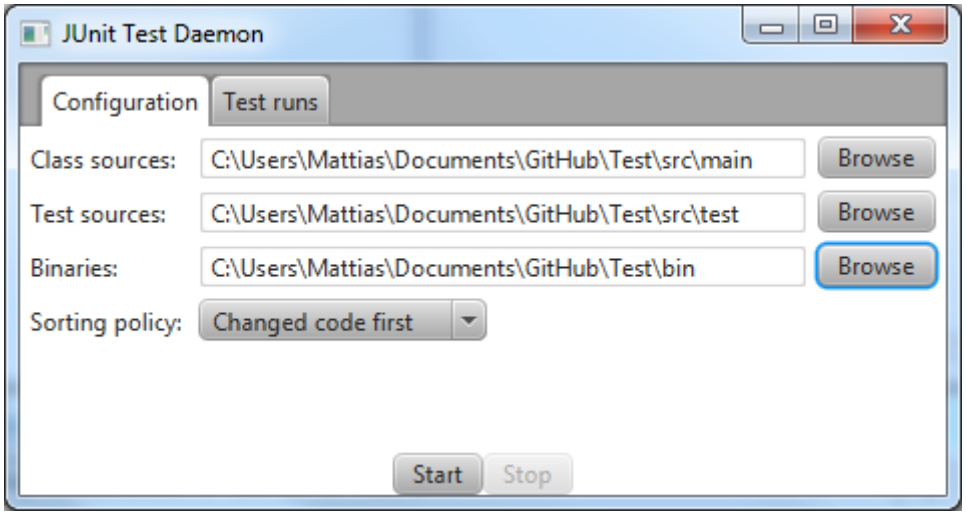
We bespreken kort elk van de negen GRASP principes waarbij we verwijzen naar ons project op een hoog niveau. Concrete voorbeelden van bijvoorbeeld High Cohesion en Low Coupling zijn impliciet besproken in de vorige ontwerpsecties en vermelden we hier niet meer expliciet. Deze GRASP patronen of principes zijn in feite triviale bouwstenen voor een goed ontwerp.

- **Controller**: Java FX, waarmee we de GUI maakten, heeft dit principe ingebouwd.



Figuur 8: Diagram van de afhankelijkheden tussen de verschillende packages, gegenereerd met STAN. De klasse `Pipeline` bevindt zich in de top-package `.group2`. De package `.group2.data` is anders voorgesteld omdat die subpackages bevat. De pijl van deze package naar `.group2.testrunner` is rood, omdat die een bidirectionele afhankelijkheid aanduid. Het gaat hier echter over het registreren van een listener. Merk op dat de package `filewatch` enkel gebruikt wordt door de package `store`.

- **Creator**: De `Pipeline` is verantwoordelijk voor het creëeren van de objecten verantwoordelijk voor de verschillende stappen. Deze laatste zullen op hun beurt objecten crëeren om de stap in kwestie tot een goed einde te brengen.
- **High Cohesion**: De beschreven klassen en deelsystemen (voor de verschillende stappen van de pipeline), zoals beschreven in de vorige secties, zijn verantwoordelijk voor één concrete taak binnen hetzelfde functionele domein (de stap van pipeline). Dit getuigt van een hoge mate van chohesie.
- **Indirection**: Het duidelijkste voorbeeld van indirecties is het veelvuldig gebruik van Listeners/Observers.
- **Information Expert**: `Test`s bevatten een collectie van `TestRun`s en zullen zelf de laatste datum van faling en het falingspercentage berekenen.
- **Low Coupling**: De koppeling tussen de verschillende packages is weergegeven in figuur 8. De voorheen besproken klassen kunnen hier eenvoudig in geplaatst worden via hun bevoegdheden. In de figuur is duidelijk een lage mate van koppeling te zien.
- **Polymorphism**: Elke besproken klassenhiërarchie is gemaakt om polymorfisme uit te buiten.
- **Protected Variations**: Als je cloudopslag wilt supporteren kan je een nieuwe `Store` subklasse implementeren. Als je een nieuwe compiler wilt gebruiken kan je een subklasse van `JavaCompiler` implementeren.
- **Pure Fabrication**: Om de koppeling laag en de cohesie hoog te houden wordt het updaten van de data in de `TestDatabase` alsook het gebruiken van de data van de `TestDatabase` niet door de `TestDatabase` zelf gedaan. Het Information Expert principe zou hier dus minder goed toegepast kunnen worden.



Figuur 9: Screenshot van het configuratiepaneel in de GUI.

1.10 Grafische gebruikersinterface

Een eenvoudige grafische gebruikersinterface werd ontwikkeld om het werken met de testdaemon eenvoudiger en overzichtelijker te maken voor de gebruiker.

De interface bestaat uit twee panelen: een configuratiepaneel en een overzichtstabel van de uitgevoerde testruns. Beide zijn weergegeven in figuren 9 en 10. In het configuratiepaneel worden de locaties van de broncode en testcode en gewenste uitvoerlocatie voor klassen ingegeven. Ook de gewenste policy kan gekozen worden. Nadat de daemon gestart is, worden de resultaten van de testruns toegevoegd aan de tabel. Meer details over een testrun (zoals de exception en stack trace van een gefaalde run) worden weergegeven door een bepaalde testrun te selecteren.

De GUI is gebouwd op het JavaFX [1] platform. Deze opvolger voor het gedateerde Swing framework biedt een MVC framework met een eigen declaratietaal voor views (FXML), een uniforme *look-and-feel* over alle besturingssystemen en geavanceerde data binding functionaliteiten om modeleigenschappen aan viewelementen te koppelen in de controller.

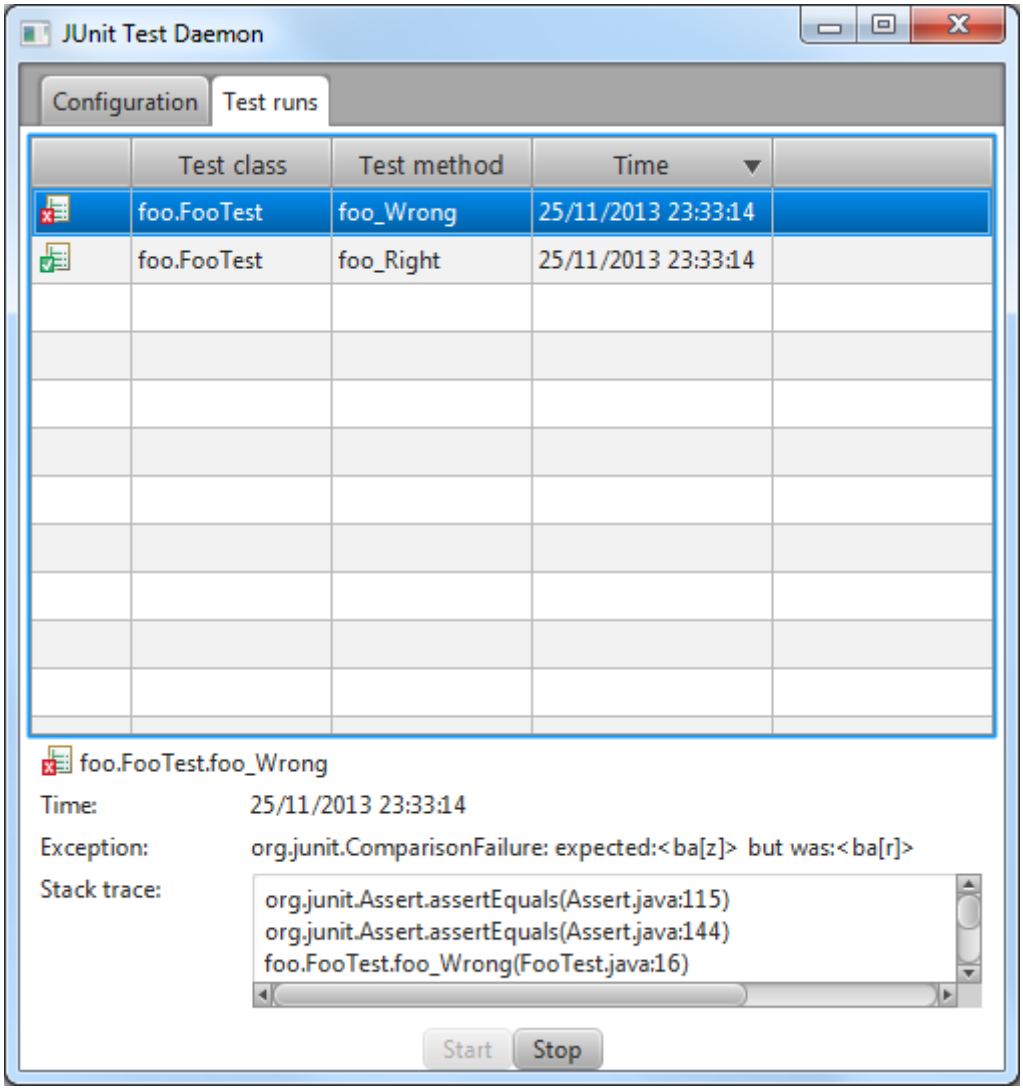
2 Testen

2.1 Hoe werd er getest?

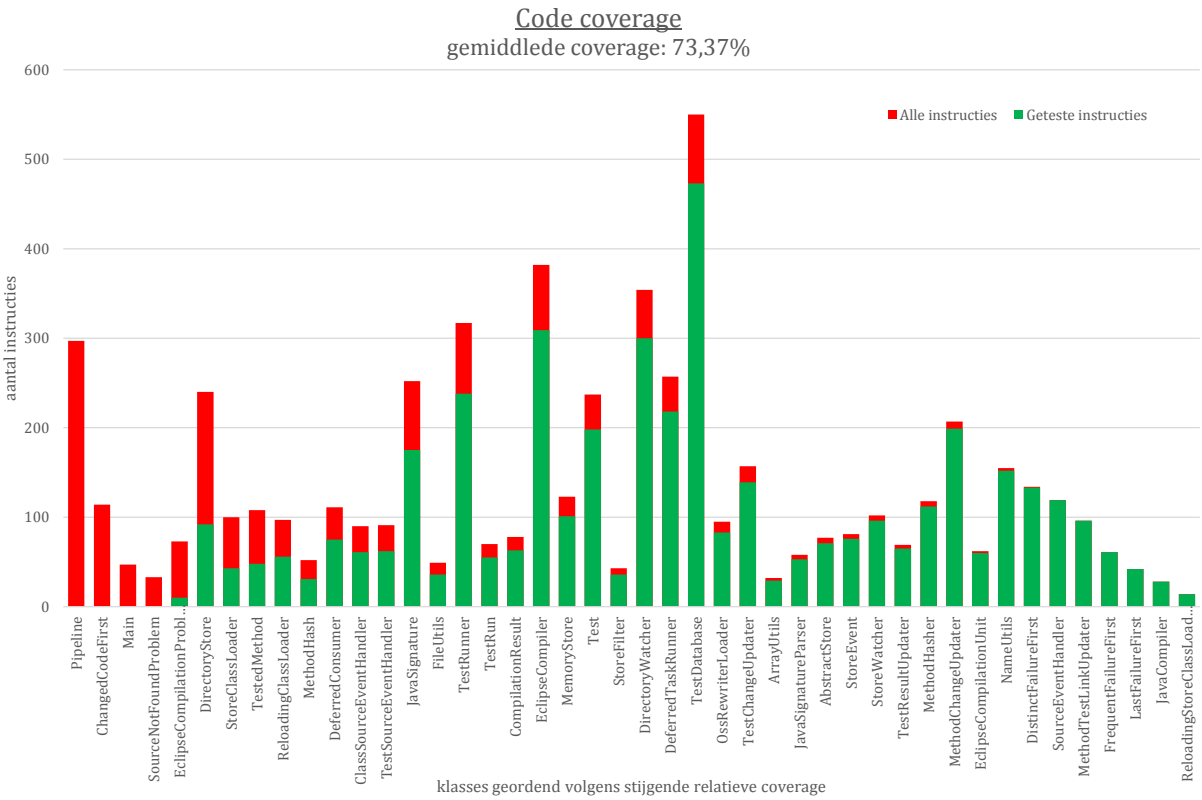
Bij het schrijven van de JUnit testen voor het project is vooral een bottom-up waterval-methodiek gevolgd. Eerst werd de code geschreven, daarna werd die geverifiëerd. Dit was mogelijk omdat het project uit veel op zich staande delen bestaat. Deze afzonderlijke delen worden dan in grotere delen gepast tot ze uiteindelijk in de pipeline worden gebruikt: de uitvoer van de vorige actie wordt ingevoerd als input van de volgende actie. Op deze manier kunnen we voor de JUnit test cases zelf de input voor de actie verzinnen en verifiëren of de output correct is.

Deze techniek werkt goed voor bijvoorbeeld de `MethodHasher` of de `JavaSignatureParser`. Daar kunnen we heel simpel concluderen dat de bugs uit deze klassen komen als de testen niet slagen, aangezien dit meer losstaande klassen zijn.

Deze techniek werkt minder goed voor bijvoorbeeld de `SourceEventHandlers` die veel meer verwoven zitten in ons project. Hier zat er namelijk tijdens het testen ook een probleem dat eigenlijk opgelost moest worden in de `EclipseCompiler` klasse. Zo werd er een ketting van testen aangemaakt om uit te vissen waar het probleem zat om dan op dat laagste niveau uiteindelijk de bug te fixen (de annotaties worden namelijk niet ondersteund bij het standaard Java source level (1.3) en ze worden niet bijgehouden voor het standaard Java target platform (1.3)). Als dit eerder werd opgevangen tijdens het testen van de `EclipseCompiler` zou er waarschijnlijk iets minder tijd gekropen zijn in het vinden van deze bugs.



Figuur 10: Screenshot van de overzichtstabel van de testruns in de GUI.



Figuur 11: Staafdiagram van de coverage van ons project, exclusief het `ui` package. Vooral het feit dat `Pipeline` niet gedekt is door unit tests haalt de gemiddelde coverage naar beneden.

2.2 Wat werd er getest?

We hebben geprobeerd om alles in ons project te testen. Dit werd geanalyseerd aan de hand van de code coverage, weergeven in een staafdiagram in figuur 11. Bij elke klasse werden enkele standaardgevallen bedacht die kunnen optreden in het normale gebruik van het programma. Een uitzondering is de topklasse `Pipeline`, die getest werd door het programma te gebruiken waarvoor het gemaakt is.

De testen zijn voornamelijk *positief*: ze testen het normale gebruik van het systeem, met normale invoer en de verwachte normale uitvoer. Er zijn niet zoveel *negatieve* testen die de respons op ongeldige invoer nagaan. Dit maakt dat de totale coverage strandt op 73% omdat de exceptionele branches niet genomen worden. Het maakt ook dat we vooral vertrouwen hebben in het systeem onder normale omstandigheden, we kunnen niet echt garanderen dat verkeerd gebruik ook altijd correct opgevangen en afgehandeld wordt.

3 Project management

Een overzicht van de gepresteerde uren en de taakverdeling is te vinden in tabel 1.

4 Verklarende woordenlijst

GoF
: Gang-of-Four, de verzamelnaam voor de vier auteurs van het handboek voor softwareontwerp *Design Patterns: Elements of Reusable Object-Oriented Software*.

GRASP
: General Responsibility Assignment Software Patterns (or Principles), een lijst van negen richtlijnen voor softwareontwerp.

GUI
Graphical User Interface, een grafische gebruikersinterface.

MVC
Model-View-Controller, een ontwerppatroon voor GUIs waarbij de controller de communicatie tussen modellen en views coördineert.

Store
De klasse in het project die resources (byte arrays) bijhoudt aan de hand van hun resource name. Deze klasse maakt abstractie van het onderliggende opslagmedium (bestandssysteem of geheugen).

Store Event
De klasse in het project die een wijziging in een Store representeert. Deze wijziging houdt zijn Store, de resourcenaam waar de wijziging is gebeurd en het type van wijziging. Dit type kan van de vorm: toevoeging, wijziging, verwijdering zijn.

Test
De klasse in het project die een testmethode representeert. Deze houdt een klasse- en methodenaam bij en de TestRuns van deze testmethode.

TestRun
De klasse in het project die een testrun representeert. Deze houdt bij of het een falende run was of niet. Ook wordt het tijdstip van de run bijgehouden en bij een falende run de stack trace.

TestedMethod
De klasse in het project die een methode in de broncode representeert. Deze houdt een signatuur, een hash en een tijdstip van laatste wijziging bij. De signatuur dient voor unieke identificatie. De hash dient om wijzigingen te detecteren.

Pipeline
De pipeline in het project is het hele gebeuren van broncode wijzigingen detecteren, deze compileren, codeveranderingen detecteren, testen sorteren en deze uiteindelijk uitvoeren. De klasse Pipeline regelt de onderdelen van de pipeline en zet ze op de juiste manier in elkaar.

Pipeline run
Een pipeline run is één volledige uitvoering van de volledige reeks operaties hierboven in Pipeline besproken.

TestDatabase
De klasse die alle data bijhoudt in verband met Tests en TestedMethods en de links tussen Tests en TestedsMethods.

Updater
Dit is een groep van klassen die de TestDatabase updaten a.d.h.v. de gedetecteerde wijzigingen in de programmacode.

Besluit

Om de gevraagde functionaliteit te realiseren delen we die op in sequentiële delen die samenwerken in een pipeline. Deze pipeline reageert vetraagd op code veranderingen om deze in bulk te behandelen. De gewijzigde code wordt gehercompileerd, wijzigingen in de testen en geteste methoden worden doorgevoerd in de database en de nieuwe lijst van testen wordt gesorteerd uitgevoerd.

In het ontwerp word rekening gehouden met de GRASP principes. De belangrijkste voorbeelden hiervan zijn de lage koppeling tussen de componenten van de pipeline en de hoge cohesie binnen die componenten. Er werd waar toepasselijk gebruik gemaakt van GoF ontwerppatronen, voornamelijk het Observer (Listener) patroon keert vaak terug.

De unit tests dekken 73% van de code. Dit komt doordat voornamelijk de meest losgekoppelde delen getest zijn en niet zozeer de samengestelde componenten zoals de pipeline. Ook zijn er niet veel negatieve testen geschreven die het exceptioneel gedrag van het systeem verifieert. We hebben dus meer vertrouwen in het systeem onder normale omstandigheden en niet zozeer onder buitengewone omstandigheden.

Referenties

[1] Oracle, *JavaFX Developer Home*. http://www.oracle.com/technetwork/java/javafx/overview/index.html, 2013.

[2] OW2 Consortium, *ASM*. http://asm.ow2.org/, 2013.

[3] The Eclipse Foundation, *JDT Core Component*. http://www.eclipse.org/jdt/core, 2013.

Tabel 1: Werkverdeling

Dag	Mattias	Vital	Ruben	Matthias	Taak
31/10	2.0	2.0		2.0	analyse opgave iteratie 2, initieel ontwerp
2/11			2.0		initieel klassendiagram, verdere analyse opgave
2/11	3.0				experimenten veranderde methode identificatie met sourcecode (AST)
3/11	2.0				experimenten veranderde methode identificatie met bytecode (ASM)
4/11	2.0	2.0	2.0	2.0	bespreking experimenten, werkverdeling, opstellen vragen voor begeleider
7/11		3.0			logging aanvullen, historyKeeper ontwerpen
7/11				2.0	TestDaemon (timer en thread pool versie), multi-threading
7/11			4.0		folderwatcher
7/11			4.0		signature parser, begin method linker
7/11		1.0	1.0		initieel diagram opstellen, nadenken over design
7/11	2.0				Compiler
9/11				1.0	Deferred run ipv periodische run / Volatiele velden
9/11	2.0				Compiler met Stores
10/11				1.0	Deferred run: Atomische operaties / Synchronisatie
10/11	4.0				DirectoryWatcher koppelen aan Store en Compiler
11/11	3.0				MethodHasher
12/11		6.0			Database en updaters, branchmanagement, overleg
13/11	2.0	2.0		2.0	Overleg
16/11			5.0	2.5	TestRunner implementeren / aan updaters werken
17/11			5.0	3.0	TestSortingPolicies implementeren / aan updaters werken
18/11				3.0	TestRuns implementeren
19/11		3.0			Database en updaters verder afwerken en testen
19/11		0.5			Coverage bepalen
20/11				3.0	TestSortingPolicies implementeren
20/11	2.0				UI: experimenteren met JavaFX
21/11		5.0	8.0	1.0	Testen schrijven, debuggen
23/11		4.0	5.0	4.0	UML diagrammen, begin verslag
23/11	10.0				UI: controllers, pipeline configuratie
24/11		3.0	4.0	6.5	Verslag, testen schrijven
24/11	6.0				UI: testresultaten
25/11	3.0	8.0	5.0	7.5	Verslag
26/11	4.5	5.5	4.5	5.5	Verslag nalezen
Totaal:	47.5	45	49.5	46	