



KU Leuven  
Departement Computerwetenschappen

# ONTWERP VAN SOFTWARESYSTEMEN

## Verslag iteratie 3

*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 derde en laatste iteratiestap spitst zich toe op het uitbreiden, refactoren, analyseren en beoordelen van het ingediende project van iteratie 2. Waarbij in iteratie 2 de nadruk lag op *Make it work* wordt er nu vooral gefocust op *Make it right*. De laatste stap *Make it fast* kwam eenmaal uitdrukkelijk aan bod bij het hertransformeren van klassen waar de performantie zichtbaar te wensen overliet.

De uitbreiding bestaat erin om testsorteringspoliticies op een eerlijke manier samen te stellen. Dit wordt geïmplementeerd met behulp van een Composite design patroon.

Het ontwerp van iteratie 2 is grotendeels hetzelfde gebleven. De sorteerfunctionaliteit van een paar testsorteringspoliticies wordt omhooggetrokken naar een nieuwe abstracte klasse. Een andere wijziging is de toevoeging van een klasse **Project**, die de drie **Stores** afschermt van de **Pipeline** om deze laatste lichter te maken. Alsook om de koppeling en cohesie met deze klassen elders in het project te verlagen respectievelijk te verhogen.

Na deze uitbreiding en refactoring wordt de code geanalyseerd met inFusion en STAN. Beide tools geven een aantal mogelijke problemen aan, die het gevolg zijn van de overwogen ontwerpkeuzes.

Tenslotte wordt ook de code coverage van de geschreven testen gecontroleerd. De gemiddelde code coverage is 80,5%.

## Inhoudsopgave

<b>Inleiding</b>	<b>2</b>
<b>1 Uitbreiding</b>	<b>2</b>
1.1 Compositie van policies . . . . .	2
1.2 Eerlijke samenstelling met round robin . . . . .	2
<b>2 Refactoring</b>	<b>4</b>
2.1 ComparingTestSortingPolicy . . . . .	4
2.2 Project en Pipeline . . . . .	4
<b>3 Eindanalyse</b>	<b>6</b>
3.1 inFusion . . . . .	6
3.2 STAN . . . . .	9
3.3 Testcoverage . . . . .	9
<b>4 Project management</b>	<b>9</b>
<b>Besluit</b>	<b>10</b>

# Inleiding

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 en uit te breiden.

In deze iteratie is er een kleine uitbreiding aangebracht aan het project van de vorige iteratie. Deze uitbreiding laat toe om testpolicy's op een eerlijke manier te combineren. Dit gebeurt aan de hand van een round robin strategie, eventueel met gewichten. Meer details zijn te vinden in sectie 1.

De resulterende code wordt gerefactored. De bestaande `TestSortingPolicy` implementaties worden herwerkt om de codeduplicatie van het sorteren van `tests` te verlagen. De `Pipeline` wordt verder opgesplitst met behulp van een `Project` om de drie `Stores` af te schermen van de `Pipeline` om deze laatste lichter te maken. Alsook om de koppeling en cohesie met deze klassen elders in het project te verlagen respectievelijk te verhogen. Deze refactoring is beschreven in sectie 2.

Daarnaast wordt het volledige project kritisch geanalyseerd. Hiervoor worden dezelfde tools gebruikt als in de eerste iteratie om het JUnit framework project te analyseren. De resultaten van deze analyse, alsook de code-coverage van de geschreven testen zijn beschreven in sectie 3.

## 1 Uitbreiding

Het eerste deel van deze iteratie bestaat erin de policy's uit te breiden opdat verschillende policy's op een eerlijke manier samengesteld kunnen worden. Dit houdt in dat iedere deelpolicy regelmatig aan de beurt komt om een volgende test te selecteren en dat geen deelpolicy altijd voorrang krijgt.

We hebben dit zelf nog uitgebreid door gewichten aan deze deelpolicy's toe te voegen. Een deelpolicy met een gewicht van 2 moet tweemaal meer aan bod komen dan een deelpolicy met een gewicht van 1 bij de samenstelling van deze deelpolicy's. Een deelpolicy met een gewicht van 2 wordt bijvoorbeeld tweemaal na elkaar geselecteerd door een `RoundRobinTestSortingPolicy`. Dit is efficiënter dan de deelpolicy tweemaal toe te voegen aan de samengestelde policy, omdat de gesorteerde lijst van het deelresultaat wordt herbruikt in plaats van opnieuw wordt berekend.

### 1.1 Compositie van policy's

Voor deze uitbreidingen worden de `CompositeTestSortingPolicy` en `WeightedTestSortingPolicy` toegevoegd aan de policyhiërarchie (figuur 1).

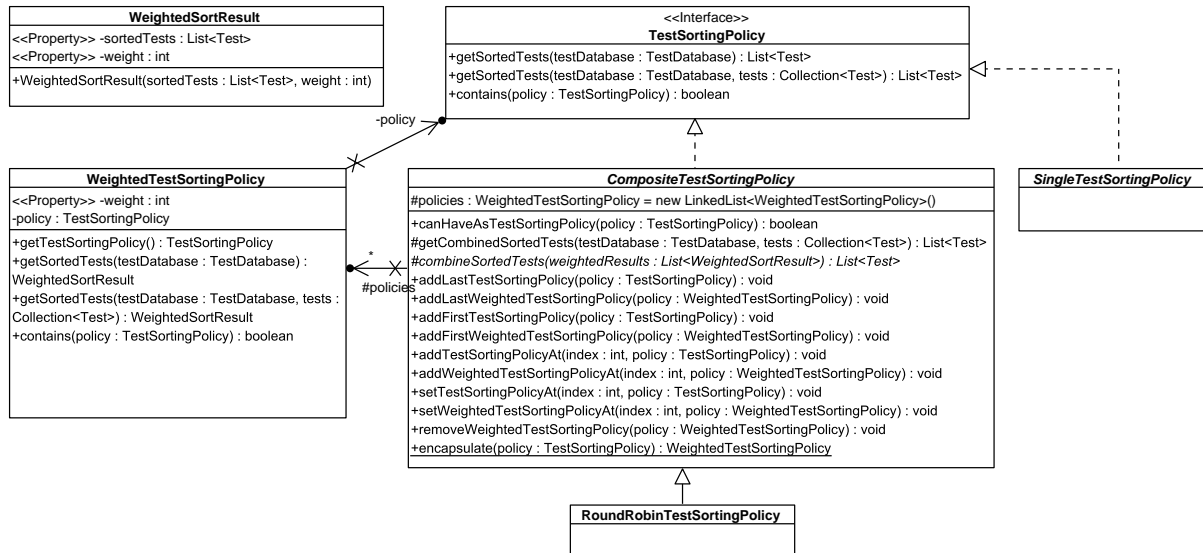
Een `CompositeTestSortingPolicy` is een compositie van `WeightedTestSortingPolicy` objecten. Het is dus geen `Composite` in de pure vorm door de indirectie via `WeightedTestSortingPolicy`. Deze abstracte klasse beheert de deelpolicy's en biedt een standaard implementatie aan voor `getSortedTests`. Daarin verzamelt het de deelresultaten van de deelpolicy's als een lijst van `WeightedSortResults` en roept dan de abstracte methode `combineSortedTests(List<WeightedSortResult>)` aan. Concrete subklassen moeten deze methode implementeren om de deelresultaten te combineren tot één gesorteerde lijst van testen.

Een `WeightedTestSortingPolicy` stelt een `TestSortingPolicy` met een gewicht voor. Het is zelf echter *geen* `TestSortingPolicy`:

- Indien het `TestSortingPolicy` zou implementeren, zou het een andere `WeightedTestSortingPolicy` kunnen bevatten en is het niet duidelijk welk gewicht in dat geval gebruikt zou moeten worden. Door `TestSortingPolicy` niet te implementeren, wordt dit probleem vermeden.
- Het is niet compatibel met de `TestSortingPolicy` interface. Het geeft zijn gesorteerde testen niet terug als een lijst van `Tests`, maar als een `WeightedSortResult`. Dit bevat de gesorteerde lijst van `Tests` alsook het gewicht van het resultaat. Op die manier kan een `WeightedTestSortingPolicy` kiezen hoe het een gewicht aan zijn resultaat geeft, standaard krijgt dit het gewicht van de `WeightedTestSortingPolicy`.

### 1.2 Eerlijke samenstelling met round robin

Een concrete implementatie van `CompositeTestSortingPolicy` wordt geleverd door de `RoundRobinTestSortingPolicy`. Deze past een round robin strategie toe op de deelresultaten. Dit is



Figuur 1: UML-klassendiagram van de **TestSortingPolicy** hiërarchie.

een eerlijke samenstelling, omdat iedere policy even vaak aan de beurt komt (afgezien van verschillen in gewichten).

In een klassieke round robin worden de delen in een circulaire buffer geplaatst en wordt het eindresultaat geconstrueerd door telkens het eerste element uit het volgende deel in de buffer te selecteren. Onze implementatie verschilt lichtjes van deze klassieke round robin:

- Alle deelresultaten bevatten dezelfde testen, maar in een andere volgorde. We moeten dus vermijden dat dezelfde test meerdere keren voorkomt in het eindresultaat. Daarom verwijderen we telkens de geselecteerde **Test** uit alle lijsten alvorens de volgende test geselecteerd wordt.
- Eenzelfde lijst mag meermaals geselecteerd worden afhankelijk van zijn gewicht. Hiervoor hebben we enkele oplossingen overwogen:
  - Iedere lijst wordt meermaals toegevoegd aan de circulaire buffer volgens zijn gewicht. Dit is echter niet zo efficiënt, omdat er geheugenruimte verloren gaat elementen (referenties) meerdere malen voorkomen en gestockeerd worden in de buffer.
  - In de lus waarin het eindresultaat wordt opgebouwd, kan een teller bijgehouden worden om na te gaan hoe vaak de huidige lijst nog hergebruikt mag worden vooraleer de volgende lijst moet geselecteerd worden. Hierbij moet wel opgelet worden dat de lus eindigt wanneer alle testen geselecteerd zijn.

Onze uiteindelijke oplossing is een combinatie van beide. De buffer is een ‘view’ in plaats van een concrete collectie: het is een concatenatie van herhaalde lijsten, opgebouwd met `Iterables.concat` en `Collections.nCopies`<sup>1</sup>. Op deze manier hebben we geen last van onnodige kopies en kunnen we nog steeds gewoon over de buffer itereren zonder zelf tellers bij te houden.

<sup>1</sup>Hoewel de naam het niet doet vermoeden, maakt `Collections.nCopies` geen *n* kopies van het gegeven element. Volgens de Javadoc is het een lichtgewicht immutable lijst dat de referentie naar het gegeven object slechts eenmaal bevat.

## 2 Refactoring

### 2.1 ComparingTestSortingPolicy

Tijdens de verdediging voor iteratie 2 werd een correct punt van kritiek aangehaald: het sorteeralgoritme van `Tests` in een `TestSortingPolicy` werd gedupliceerd in verschillende concrete subklassen van `TestSortingPolicy`. Dit zou beter gecondenseerd worden op één enkele locatie met een `Comparator<Test>` die de subklassen dan aanreiken.

Concreet hadden twee van de vier `TestSortingPolicies` (`FrequentFailureFirst` en `LastFailureFirst`) een identieke implementatie voor het sorteren met behulp van een `Comparator<Test>`. `ChangedCodeFirst` maakte gebruik van een inwendige `Tuple` klasse die `Comparable<Tuple>` implementeerde. De data waarop gesorteerd moest worden, kon niet rechtstreeks uit de `TestDatabase` of de `Test` gehaald worden, dus werd ze op voorhand opgehaald en in `Tuples` (een inwendige klasse van de `ChangedCodeFirst` klasse) opgeslagen alvorens te sorteren. De `Tests` alsook hun ordening werden vervolgens terug opgehaald uit de gesorteerde collectie van `Tuples`.

In de nieuwe versie implementeert `ChangedCodeFirst` ook `Comparator<Test>`. In plaats van de gegevens op voorhand op te vragen, worden ze herberekend bij iedere vergelijking. Dit is in principe trager, maar zorgt wel voor een duidelijker ontwerp. Deze drie policies hebben nu een gemeenschappelijke superklasse `ComparingTestSortingPolicy` gekregen. De subklassen zijn verplicht een `Comparator<? super Test>` object terug te kunnen geven. Deze `Comparator` wordt dan gebruikt om de `Tests` te sorteren in de `getSortedTests` implementatie van de `ComparingTestSortingPolicy`. De drie policies hoeven dus enkel nog een `Comparator` te leveren.

De `DistinctFailureFirst` policy leent zich niet voor het implementeren van een `Comparator<Test>`. `DistinctFailureFirst` policy houdt namelijk een toestand bij die wijzigt tijdens het sorteren en afhangt van wat al gesorteerd is. Er wordt tijdens het sorteren drie collecties bijgehouden in een `DistinctFailureFirst` policy:

- Een collectie die de `Tests` met de hoogste prioriteit bevat. Deze geordende collectie vormt het eerste deel van de uiteindelijk teruggegeven geordende `Tests`.
- Een collectie die de `Tests` met de laagste prioriteit bevat. Deze geordende collectie vormt het tweede deel van de uiteindelijke teruggegeven geordende `Tests`.
- Een collectie met de `StackTraceElements` van de oorzaken van falen die reeds tegengekomen zijn.

`Tests` die een `TestRun` bevatten die een nieuwe oorzaak van falen oplevert, worden gestockeerd in de collectie van de `Tests` met de hoogste prioriteit. De oorzaak wordt bijgehouden in de collectie van `StackTraceElements`, opdat volgende `Tests` met dezelfde oorzaak niet meer de hoogste prioriteit krijgen. Indien de `Test` geen enkele `TestRun` bevat die een nieuwe oorzaak van falen oplevert, krijgt de `Test` de laagste prioriteit.

Het klassendiagram van deze vier `SingleTestSortingPolicies` is weergegeven in figuur 2. Deze hiërarchie kan eenvoudig ingepast worden in de bovenliggende hiërarchie weergegeven in figuur 1.

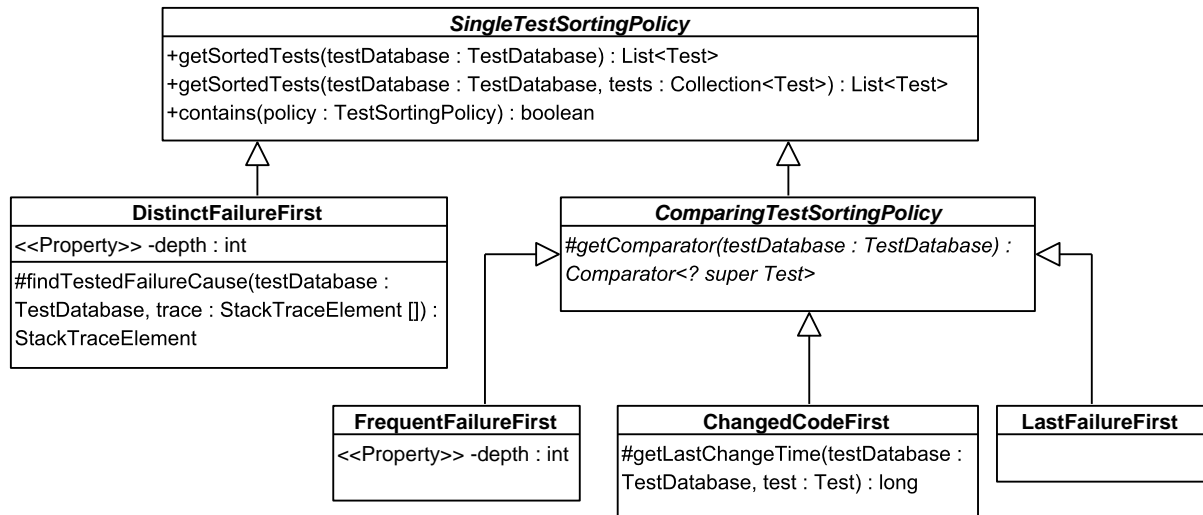
### 2.2 Project en Pipeline

Bij het bespreken van onze (vooruit)gang voor iteratie 3 met de assistent, werd de grote constructor van de `Pipeline` klasse als complex ervaren. De `Pipeline` is verantwoordelijk voor het creëren en het koppelen (bij initialisatie) van een aantal verschillende componenten. Vervolgens moet de `Pipeline` zich niet meer wat betreft de onderlinge communicatie tussen deze componenten.

Om het overzicht te behouden alsook om de koppeling te verlagen tussen sommige klassen wordt er een `Project` klasse geïntroduceerd. Het `Project` neemt een deel van de taken van de originele `Pipeline` over, zodat de `Pipeline` niet altijd rechtstreeks met de afzonderlijke componenten moet werken. Dit is een toepassing van het Pure Fabrication GRASP principe: het stelt niet echt een model uit het probleemdomein voor, maar dient enkel om lagere koppeling en hogere cohesie te bekomen.

De `Project` klasse bevat de volgende methoden:

- `Project(Store classSourceStore, Store testSourceStore, Store binaryStore)`: Deze constructor ontvangt drie `Store` objecten: een voor de broncode van niet-testcode (`classSourceStore`),



Figuur 2: Klassendiagram van de gerefactorde `TestSortingPolicy` hiërarchie.

een voor de broncode van testcode (`testSourceStore`) en een voor alle gecompileerde code (zowel test- als niet-testcode) (`binaryStore`). Voor deze eerste twee `Stores` wordt een `StoreWatcher` aangemaakt. Verder wordt er ook een `ReloadingStoreClassLoader` op basis van de `binaryStore` aangemaakt.

- `startListening(Consumer<StoreEvent> consumer)`: Beide `StoreWatchers` registreren de meegegeven `Consumer`. De `Stores` registreren de corresponderende `StoreWatcher` en starten verandering in de code door te sturen naar hun `StoreListeners`.
- `stopListening(Consumer<StoreEvent> consumer)`: Beide `StoreWatchers` ontregistreren de meegegeven `Consumer`. De `Stores` ontregistreren de corresponderende `StoreWatcher` en stoppen verandering in de code door te sturen naar hun `StoreListeners`.
- `getClassLoader()`: Deze methode geeft de `ReloadingStoreClassLoader` terug, dewelke o.a. gebruikt wordt in de `TestRunner`.
- `isBinaryClass(String className)`: Deze methode controleert of de meegegeven klassennaam correspondeert met een gecompileerde klasse van de `binaryStore`. Deze methode wordt gebruikt door de `Pipeline` om een filter (`Predicate<String>`) te maken voor de `OSSRewriter`, opdat deze enkel de gecompileerde klassen van de `binaryStore` herschrijft. In iteratie 2 maakten we niet gebruik van een filter waardoor `retransformAllClasses()` aanzienlijk lang duurde. Dit heeft te maken met het enorme aantal klassen van de bibliotheken van derden (Guava, JavaFX,...) alsook onze eigen klassen en JUnit klassen die meegetransformeerd moeten worden. De filter kan de te transformeren gecompileerde klassen tot een minimum beperken.
- `createClassCompiler()`: Deze methode maakt een `JavaCompiler` gebaseerd op de `classSourceStore`, `binaryStore` en `classLoader` van het `Project`. Door deze methode te gebruiken in de `ClassSourceEventHandler` wordt de koppeling vermindert doordat deze laatste enkel aan de `classSourceStore` van het `Project` en een `JavaCompiler` via het `Project` moet geraken. Dit verhoogd eveneens de cohesie. De `ClassSourceEventHandler` moet nu zelf niet meer weten hoe of welke `JavaCompiler` te initialiseren. Daarnaast moet deze helemaal niets weten van het bestaan van de `binaryStore` en `classLoader`.
- `createTestCompiler()`: Deze methode maakt een `JavaCompiler` gebaseerd op de `testSourceStore`, `binaryStore` en `classLoader` van het `Project`. Dit is analoog aan `createClassCompiler()` maar dan voor de `TestSourceEventHandler`.



Figuur 3: De overview pyramid van ons project. We zien dat ongeveer alle metrieken minder dan gemiddeld zijn.

### 3 Eindanalyse

Ons uiteindelijke project wordt onderworpen aan twee analysetools uit de eerste iteratie: inFusion en STAN. Er wordt ook gekeken naar de code-coverage van de geschreven testen.

#### 3.1 inFusion

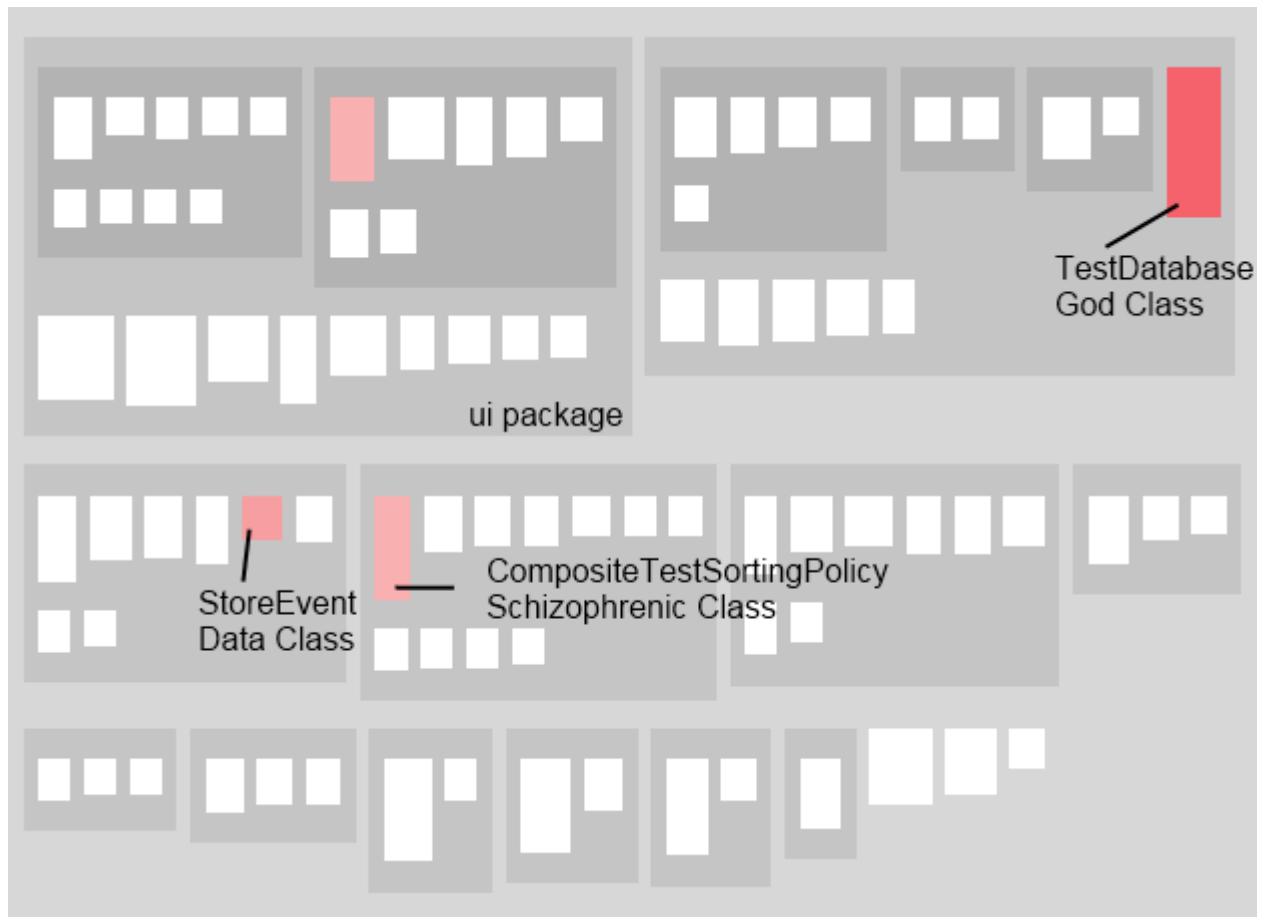
De overview pyramide is weergegeven in figuur 3. We kunnen zien dat het project voor veel van de criteria minder dan gemiddeld scoort. O.a. het aantal methodes per klasse, het aantal lijnen code per klasse, etc.. Er zijn drie uitzonderingen: de cyclometrische complexiteit en aangeroepen methodes zijn scores gemiddeld. Daarnaast zijn de hiërarchiën dieper dan gemiddeld.

Daarnaast rapporteert inFusion nog enkele design flaws:

- **StoreEvent** zou een dataklasse zijn. Dit is een klasse die zelf weinig functionaliteit heeft. Wij vinden dit echter in dit geval niet echt een probleem aangezien **StoreEvent** ook effectief een soort record is dat verstuurd wordt, opdat de data erin gebruikt kan worden. Figuur 4 geeft de design flaw view weer.
- **TestDatabase** zou een godklasse zijn. Deze klasse heeft inderdaad een groot aantal methodes omdat dit het centraal aanspreekpunt is voor het opslaan van de data. Deze data is sterk met elkaar gerelateerd en er is dus voor gekozen om de cohesie van deze klasse hoog te houden door dit bij elkaar te plaatsen. Er is wel geprobeerd om dit deels op te lossen door **Updater** klassen te gebruiken die ervoor zorgen dat de database op de juiste manier geüpdatet wordt.
- **CompositeTestSortingPolicy** zou een schizofrene klasse zijn. Deze klasse heeft namelijk een heleboel methoden om de collectie van **WeightedTestSortingPolicies** aan te passen.

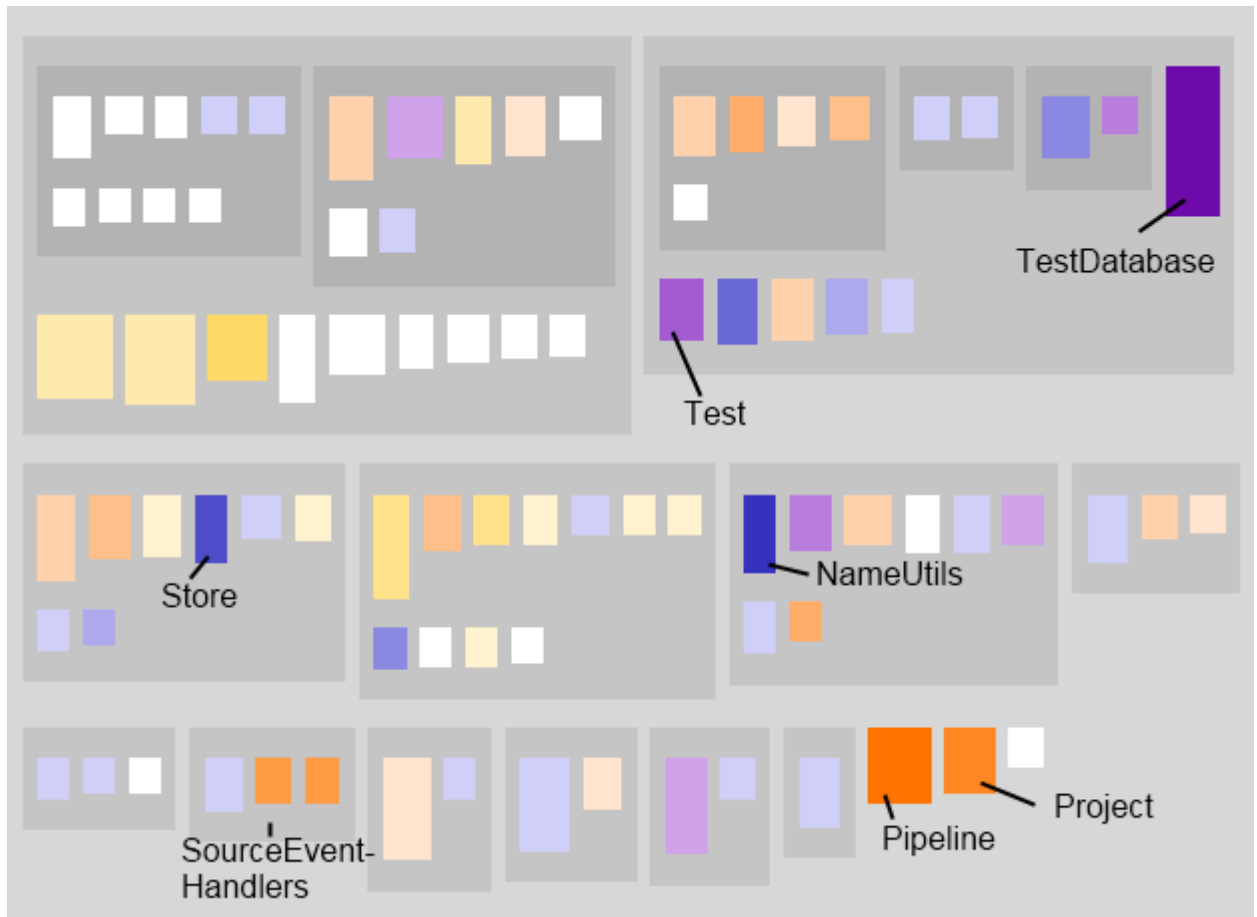
Deze design flaws zijn dus niet echt design flaws maar een gevolg van zorgvuldig overwogen ontwerp-beslissingen.

Figuur 5 geeft de coupling view weer. De paarse klassen zijn klassen die vaker gebruikt worden dan dat zij anderen gebruiken. De oranje klassen zijn klassen die vaker andere klassen gebruiken dan dat zij gebruikt worden. Enkele klassen die donkerder van kleur zijn, en dus sterker gekoppeld, zijn benoemd. Enkele interessante gevallen zijn bijvoorbeeld **TestDatabase**, **Test** en **Store** die een sterke ‘providers’ zijn van methodes. Daarnaast zijn **Pipeline**, **Project** en de **SourceEventHandlers** sterke ‘users’ van externe methodes.

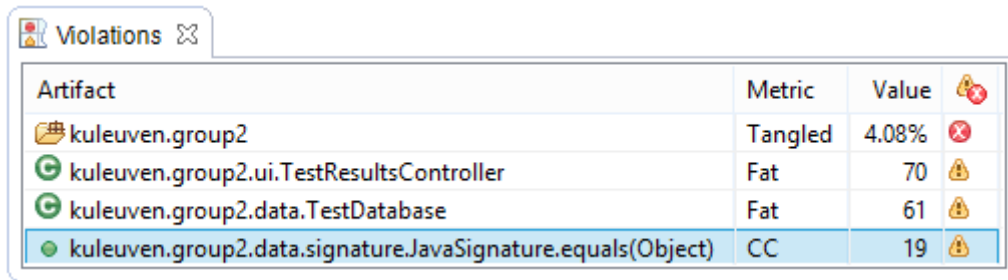


Figuur 4: InFusion design flaw view. Er zijn drie design flaws te zien. TestDatabase is een godklasse, StoreEvent een dataklasse en CompositeTestSortingPolicy een schizofrene klasse.





Figuur 5: De koppeling in het project. De paarsere klassen zijn 'providers' van methodes. De oranjere klassen zijn 'users' van methodes. Enkele interessante gevallen zijn benoemd.



Artifact	Metric	Value	
kuleuven.group2	Tangled	4.08%	✖
kuleuven.group2.ui.TestResultsController	Fat	70	⚠
kuleuven.group2.data.TestDatabase	Fat	61	⚠
kuleuven.group2.data.signature.JavaSignature.equals(Object)	CC	19	⚠

Figuur 6: Violations gerapporteerd door STAN.

We concluderen dat inFusion vooral positieve feedback geeft over het project. De resultaten liggen in lijn met onze verwachtingen en de opgemerkte flaws zijn gevolgen van onze ontwerpbeslissingen.

### 3.2 STAN

STAN rapporteert vier violations. De grafische output staat in figuur 6. Deze violations kunnen worden verklaard:

- Tangledness van `kuleuven.group2` : meer specifiek gaat het hier over de packages `data.updating` en `testrunner`. De `TestRunner` gebruikt de klasse `Test` (8 afhankelijkheden), en de `MethodTestLinkUpdater` luistert naar de `TestRunner` (2 afhankelijkheden). Daardoor zijn de packages afhankelijk van elkaar, en is er dus inderdaad een (kleine) tangle.
- Fatness van `TestResultController` : deze klasse is een onderdeel van de GUI-code. Dit valt buiten de focus van het ontwerp.
- Fatness van `TestDatabase` : deze klasse bevat inderdaad veel methodes (61). Dit zijn voornamelijk kleine methodes (`addX`, `getX`, `containsX`, etc.) voor de `Tests`, `TestRuns`, `TestBatches` en `MethodTestLinks`. Er zijn ook een paar methodes om de `TestDatabaseListeners` in te lichten (`fireX`). Zoals reeds uitgelegd in sectie 3.1 is dit een ontwerpkeuze: de opgeslagen data is sterk gerelateerd en we verkiezen hier hogere cohesie.
- Cyclometrische complexiteit van `JavaSignature.equals` : een `equals` methode met veel (mogelijk `null`) velden heeft uiteraard veel `if`-statements. Daardoor zijn er veel mogelijke paden door de methode, wat resulteert in een hoge cyclometrische complexiteit.

We besluiten dat de kritieken van STAN te verklaren zijn binnen ons ontwerp; ze worden niet gecorrigeerd.

### 3.3 Testcoverage

De testen uit de vorige iteratie worden uitgebreid. Onveranderde code wordt strenger getest en nieuwe code wordt ook getest.

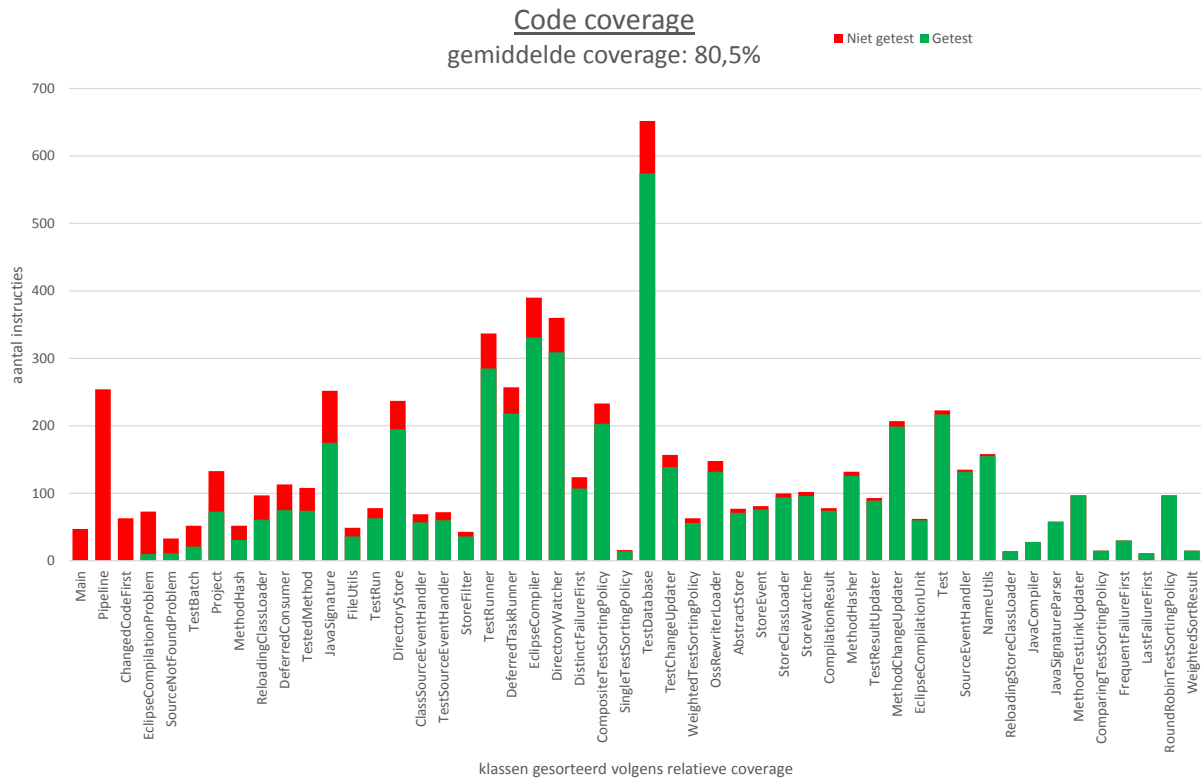
Een overzicht van de code-coverage is te zien in 7. De gemiddelde code-coverage is iets verbeterd (80,5%) ten opzichte van de vorige iteratie (73,37%).

Ook is er iets meer aandacht gespendeerd aan het negatief testen. Dit was namelijk minder het geval in de vorige iteratie. De exceptionele branches worden nu meer getest en de correcte afhandeling wordt geverifieerd.

De uitzondering van de testen is het `ui` package. Dit is complexer, omdat de gebruikersinteractie moet gesimuleerd worden en de juiste uitvoer moet gecontroleerd worden. Er is even gezocht naar mogelijkheden om de UI te testen, zoals `MarvinFX` en `TestFX`. Omdat de GUI buiten de focus van dit vak valt, werd gekozen om de GUI niet te testen.

## 4 Project management

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



Figuur 7: Staafdiagram van de testcoverage per klasse.

## Besluit

De uitbreiding van het project uit iteratie 2 kon vrij eenvoudig geïmplementeerd worden. Daarnaast is de policy-hiërarchie aangepast om te beantwoorden aan een opmerking tijdens de laatste verdediging. De **Pipeline** klasse wordt een klein beetje verlicht door de **Stores** op te nemen in een **Project** klasse.

Ten slotte wordt de bekomen code geanalyseerd met inFusion en STAN. Beide tools geven wat commentaar, maar dit bleek niet zo relevant te zijn. De gevonden flaws zijn niet echt fouten maar eerder gevolgen van bewuste ontwerpkeuzes. De code coverage (gemiddeld 80.5%) van de testen is behoorlijk om een goed vertrouwen in het systeem te hebben.

Tabel 1: Werkverdeling

Dag	Mattias	Vital	Ruben	Matthias	Taak
4/12	4.5	4.5	4.5	4.5	bespreken opgave/testen
10/12	3.0	3.0	3.0		extra functionaliteit/testen/branch mergen
12/12	4.5	2.5	4.0	4.5	extra functionaliteit/testen
13/12	1.5		1.5	1.5	bespreking iteratie 2 en 3 met assistent/refactor mogelijkheden bespreken
14/12			2.0	1.0	testen/ui testen opzoeken en proberen
14/12	3.0				UI: Composite policies ontwerpen
15/12	1.0			1.0	bespreking TestSortingPolicies
16/12				2.5	TestSortingPolicies: refactoring/testen/bug fixes
19/12	9.0	9.0	9.0	9.0	branches mergen, Stores in Project steken, verslag
Totaal:	26.5	19.0	24.0	24.0	