KU LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN

Ontwerp van Softwaresystemen

Report: iteration 3

Joren Verspeurt (r0258417)
Sophie Marien (s0216517)
Stef Noten (s0211264)
Toon Nolten (r0258654)
Begeleider: Mario H. C. T.

Computer Science 2013 – 2014

Contents

Introduction

In the first iteration we assessed the design and implementation of *JUnit* and in the second iteration we extended it with a daemon that would continuously test a given project according to a policy chosen by the user. In this iteration we have to expand our `Daemon` to allow a combination of multiple policies. The combination of policies has to be fair. After the expansion we have to refactor the whole project and analyse it like we did in iteration 1.

In section 1 the policies are discussed, the design for the extension with the implementation of the composition of policies and the combination of policies, the merging sortings, are explained. In section 2 the overall design, the refactoring and the analysis of our extension of *JUnit* are discussed. In the final section (3) we discuss and evaluate the project.

1 Activity 1: support combined policies

1.1 Policies: design before the extension

In the current design, policies are represented by the `IPolicy` interface. As can be seen in figure 1, this interface exposes the method `apply`, which transforms a *JUnit* `Request` into a `Request` that complies with the properties of the policy.

In the assignment of iteration 2, all the specified policies are sorting policies. The root class for all sorting policies is the `SortingPolicy` class. Its `apply` method thus transforms a given `Request` into a `Request` of which the tests are sorted according to the policy's rules. It does this by using the `getComparator` method that all concrete `SortingPolicys` must implement.

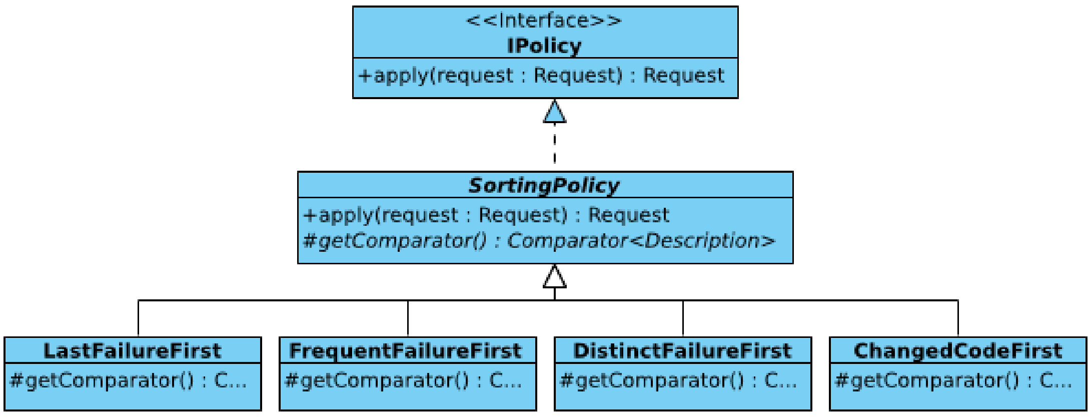
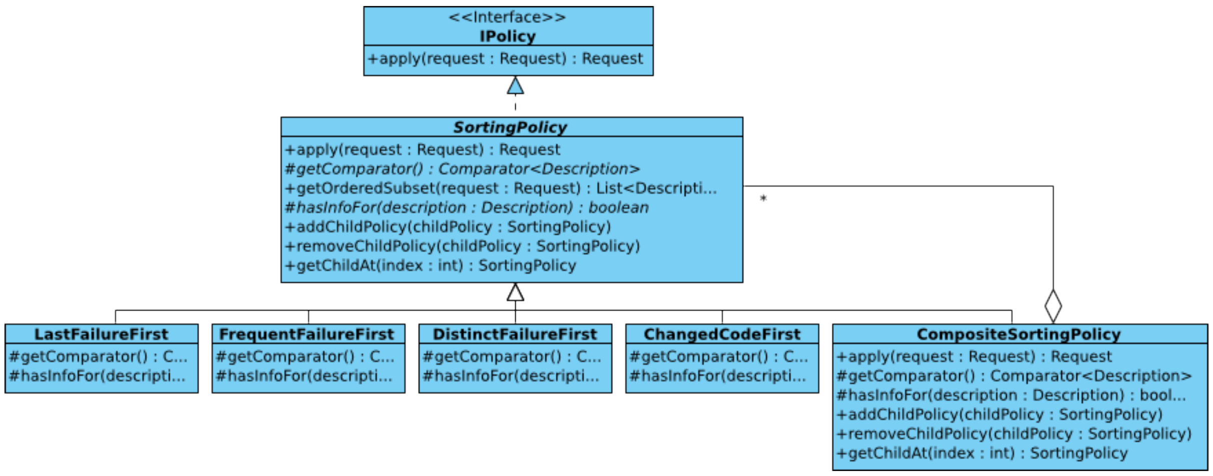


Figure 1: Class diagram of policies before the extension

1.2 Design with the extension

It should be possible to combine sorting policies into a new sorting policy. These combined policies should be treated in the same way as other policies. For this, the Composite pattern is used. As can be seen in figure 2, the composition is made at the level of the `SortingPolicy` class instead of the toplevel `IPolicy` interface because other policies could be defined that do something else than sorting and the combination described in the assignment only makes sense for sorting policies.

The Component of the Composite pattern corresponds with the `SortingPolicy` class, the Composite is the new class `CompositeSortingPolicy` and the Leafs are the other policies that already existed. Consequently, instances of the new `CompositeSortingPolicy` can be treated like a normal `SortingPolicy` and `IPolicy`. Composition of sorting policies is not exposed through `IPolicy`. However, the `SortingPolicy`



<<Interface>>
IPolicy
+apply(request : Request) : Request
SortingPolicy
+apply(request : Request) : Request
#getComparator() : Comparator<Description>
+getOrderedSubset(request : Request) : List<Descripti...
#hasInfoFor(description : Description) : boolean
+addChildPolicy(childPolicy : SortingPolicy)
+removeChildPolicy(childPolicy : SortingPolicy)
+getChildAt(index : int) : SortingPolicy
*
LastFailureFirst
#getComparator() : C...
#hasInfoFor(descripti...
FrequentFailureFirst
#getComparator() : C...
#hasInfoFor(descripti...
DistinctFailureFirst
#getComparator() : C...
#hasInfoFor(descripti...
ChangedCodeFirst
#getComparator() : C...
#hasInfoFor(descripti...
CompositeSortingPolicy
+apply(request : Request) : Request
#getComparator() : Comparator<Description>
#hasInfoFor(description : Description) : bool...
+addChildPolicy(childPolicy : SortingPolicy)
+removeChildPolicy(childPolicy : SortingPolicy)
+getChildAt(index : int) : SortingPolicy

Figure 2: Class diagram of policies with the extension

class exposes methods for adding, removing and retrieving a child policy. It implements default behaviour for these methods: `getChildAt` returns null, `addChildPolicy` and `removeChildPolicy` throw exceptions. Leaf policies inherit this default behaviour, but the `CompositeSortingPolicy` class overrides these methods with the expected behaviour: it actually adds, removes and retrieves the child policies.

A fair combined order

What remains is to determine the composed order in a fair way. Fairness, according to the assignment, corresponds to the concept of fairness in the context of process scheduling. The chosen approach is to let the child policies take turns choosing the next test to be run. However, the ordering can be made a little more useful. The problem with taking turns over the complete ordering of each child policy is that some policies could run out of important tests so that other, more important tests have to wait.

For example, the goal of the frequent failure first policy is to make sure frequently failed tests are executed earlier. However, some tests have a zero failure count and should execute at the end of the testrun. The same is true for tests that test unchanged code under the changed code first policy.

Thus, when a policy with unimportant tests at the end is combined with another policy that has more important tests, the execution eventually takes turns between important tests and unimportant tests (those with a zero failure count, those that execute unchanged code, etc.). In this case, it would make more sense to postpone the execution of these unimportant tests until the important tests have been executed.

Design for the fair combined order

The discussed improvement requires a change in the current design, since policies should now be able to mark tests as being important or not. For this, concrete policies should implement the `hasInfoFor` method. When a policy has useful info for a test (a return value of true), it is an important test. When it does not have useful info for a test (a return value of false), the test is not important according to the policy and its execution should be postponed until the end of the testrun.

With this improvement, a combined policy will let its child policies take turns until they run out of important tests. After this, they are skipped in the turns. When an order is determined for all the important tests, all the unimportant tests are put at the end in an undetermined order, since they have the same priority.

It is not the responsibility of a composite policy to determine the ordered list of tests of its child policies. This is the responsibility of the child policy itself. For this reason, the `getOrderedSubset` method is introduced on the `SortingPolicy` class. This method returns an ordered list of the tests for which it has info, so the unimportant tests are not included. To determine its own order, a composite policy merges the ordered lists that it retrieves from its child policies. The comparator that is returned by its `getComparator` method distinguishes between three cases:

- Both tests are unimportant: comparison renders them equal (a value of 0).
- One of the two tests is unimportant: comparison gives the unimportant one a lower priority (a negative value).
- None of both are unimportant: comparison gives higher priority to the test that occurs earlier in the merged list (a positive value).

In this way, unimportant tests are executed at the end of the testrun, as desired.

The merging algorithm will be discussed in detail in the next section.

1.3 Combination of policies: merging sortings

A composite sorting policy retrieves a sorted list of tests from each of its child policies. To determine the order imposed by the composite policy, these lists are merged. The merging algorithm is implemented as follows:

1. Loop over the sorted lists of the child policies.
2. Take the first test from the current list and remove it from all lists.
3. Repeat until all lists are empty.

Table 1 gives an example of this merging process. The tests in the request to be ordered are A to I. If a policy has no info about a test (the test thus is unimportant according to it), it is marked with an apostrophe ('). These tests are not in the sorted list that is retrieved from the child policies. When a policy's list becomes empty in the algorithm, it will not be allowed a turn any more. Tests that have not occurred in any of the lists are ordered last with an undefined order (test I' in the example).

`Policy 1`	`Policy 2`	`Policy 3`	`Policy 4`	Order
A	B	A	E	A
B	D	B	D	B
C	E	C	G	C
D	G	F	A'	E
E	F	D'	C'	D
F	H'	E'	A'	G
G	A'	G'	J'	F
H	C'	H'	H'	H
I'	J'	I'	I'	I' or J'
J'	I'	J'	F'	J' or I'

Table 1: Example of the merging of policies

2 Activity 2: refactoring and analysis

2.1 Current overall design

In the current design, `DataCollectors` collect data for tests. Implementations are present that collect test failures, dependencies of tests and code changes. Each concrete `DataCollector` collects a specific class of data instances (which implements the `ITestData` interface). Each instance of a concrete `ITestData` corresponds with a certain test (an atomic test, a suite, etc.). When a party is interested in a certain kind of `ITestData`, it must implement the `IDataCollectedListener` interface and register the listener with an `IDataEnroller` for that class of `ITestData`. The only implemented concrete `IDataEnroller` internally directs this subscription to a concrete `DataCollector` instance that can produce the requested kind of `ITestData`. It also removes the need of creating multiple data collectors that collect the same kind of data.

The raw data that is collected by `DataCollectors` is summarized by `Statistics`. Implementations are available for failure counts, points of failure, dates of the last failures and dates of the last dependency changes. Each concrete `Statistic` summarizes to a specific class of `ITestStatistics`, each corresponding with a certain test (again: an atomic test, a suite, etc.). On a `Statistic`, that concrete `ITestStatistic` can be requested for a certain test. When a party is interested in a certain concrete class of `ITestStatistic`, it can request a `Statistic` for that kind of `ITestStatistic` from an `IStatisticProvider`. An `IStatisticProvider` shields the concrete `Statistic` implementations from the user and allows sharing a `Statistic`. This way, only one `Statistic` is summarizing the raw collected data, instead of one instance per interested party.

An `IPolicy` is responsible for transforming a certain *JUnit* `Request` into a `Request` that complies with the properties of the policy. Only sorting policies are implemented, which inherit from the abstract `SortingPolicy` class. All implemented leaf policies retrieve the data on which their sorting is based from a `Statistic`. They thus retrieve this `Statistic` from an `IStatisticProvider`.

The `Daemon` class was responsible for actually running the tests under a certain policy. Together with this, it handled the initialization: it created policies, a `DataEnroller`, a `StatisticProvider` and a `RunNotifier`. Additionally, it created a `RunNotificationSubscriber`, which is a protection proxy for a `RunNotifier`. This class only exposes methods for adding and removing a `RunListener`, as opposed to including methods for firing events. As is discussed in the following section, `Daemon` had to many responsibilities.

The `ConsoleView` class notifies the user of the results of tests and testruns and allows for user interaction, such as configuring the policy or queueing a new testrun.

2.2 Refactoring

Two mayor refactorings were done: the functionality of `Daemon` and `ConsoleView` was split into multiple classes that they are more coherent. Both classes had too much responsibilities and were not as coherent as they should be.

Along these bigger refactorings, another smaller refactoring was done: some classes were moved to other packages because they belonged better there and because it improved modularity.

These bigger refactorings are discussed in the next sections.

2.2.1 Daemon refactoring

`Daemon` was responsible for repeatedly running a collection of tests as well as creating everything needed by policies. This includes a `RunNotifier` which notifies listeners of events while running tests. Additionally, it created a `DataEnroller`, a `StatisticProvider` and a `RunNotificationSubscriber`. The class diagram of the refactoring can be seen in figure 3.

The `Daemon` class was meant as a facade but it contained too much functionality. As such, it was not as coherent as it should be and it was coupled with constructors and methods of a lot of classes. For example testruns were handled manually by `Daemon` , which included loading the root suite class with a class loader, creating a `Request` for it and firing events for the start and end of a testrun. Our solution was to extract the initialization and configuration functionality into the new `DaemonSystem` class. Additionally, the `TestRun` and `TestRunCreator` classes were introduced. This is because the domain concept of a testrun was not clearly separated in code in the previous design.

The newly introduced DaemonSystem class is responsible for creating the necessary objects and doing the necessary configuration for starting a Daemon instance. Daemon previously was meant as the facade to our extension of *JUnit* , but it was not a pure facade, since it implemented too much functionality itself. The DaemonSystem class now has taken over the facade functionality, which is its only responsibility. It only exposes methods to:

- configure the registered policies,
- add or remove a RunListener (to be informed of events and results),
- start and stop the test daemon,
- queue new testruns.

The responsibility for running the tests in a certain Request was abstracted into TestRun. It is also responsible for firing events that indicate the start and end of a testrun. TestRuns can be created according to a specific IPolicy by using the TestRunCreator class. Doing the instantiation in this way decouples TestRun from IPolicy in a way that maintains its generality. This means that a TestRun represents a general testrun, independent of whether a policy is used or not. The TestRunCreator also decouples Daemon from the creation of TestRun instances.

In figure 4 the sequence diagram of the refactored startup is shown. The ConsoleView starts the DaemonSystem, which starts the Daemon . The Daemon then starts running TestRuns in a loop. While running tests, the ConsoleView waits for input from the user. Only if the user indicates that he wants to stop, the DaemonSystem, and consequently the Daemon, is stopped.

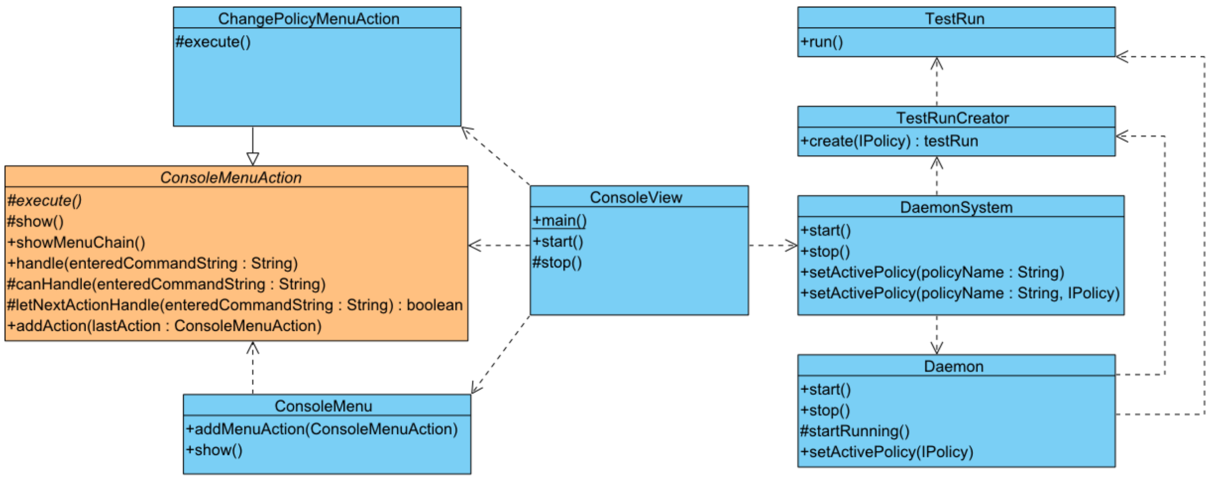

ChangePolicyMenuAction
#execute()
ConsoleMenuAction
#execute()
#show()
+showMenuChain()
+handle(enteredCommandString : String)
#canHandle(enteredCommandString : String)
#letNextActionHandle(enteredCommandString : String) : boolean
+addAction(lastAction : ConsoleMenuAction)
ConsoleMenu
+addMenuAction(ConsoleMenuAction)
+show()
ConsoleView
+main()
+start()
#stop()
TestRun
+run()
TestRunCreator
+create(IPolicy) : testRun
DaemonSystem
+start()
+stop()
+setActivePolicy(policyName : String)
+setActivePolicy(policyName : String, IPolicy)
Daemon
+start()
+stop()
#startRunning()
+setActivePolicy(IPolicy)

Figure 3: Classdiagram of the refactored part of the project

2.2.2 ConsoleView refactoring

Another refactoring was done in the ConsoleView class. ConsoleView implemented a lot of functionality, including outputting menus and letting the user choose one of the menu items. This menu handling was not implemented in a modular and reusable way, which was even more important since new menus had to be added for creating composite policies. This functionality therefore was extracted into additional classes: ConsoleMenu, ConsoleMenuAction and ChangePolicyMenuAction. Together with the ConsoleView class, these classes are placed in the new package cli (command line interface).

The ConsoleMenuAction class represents an item in a console menu with a corresponding action that needs to be executed when the item is selected. To show it in the menu, it needs to have a title. Additionally, it owns a command string which the user should enter in the console to choose the menu item and execute its action. For example, when the user enters *bla*, the action of the ConsoleMenuAction with the command string *bla* must be executed. This mechanism is implemented with the Chain of

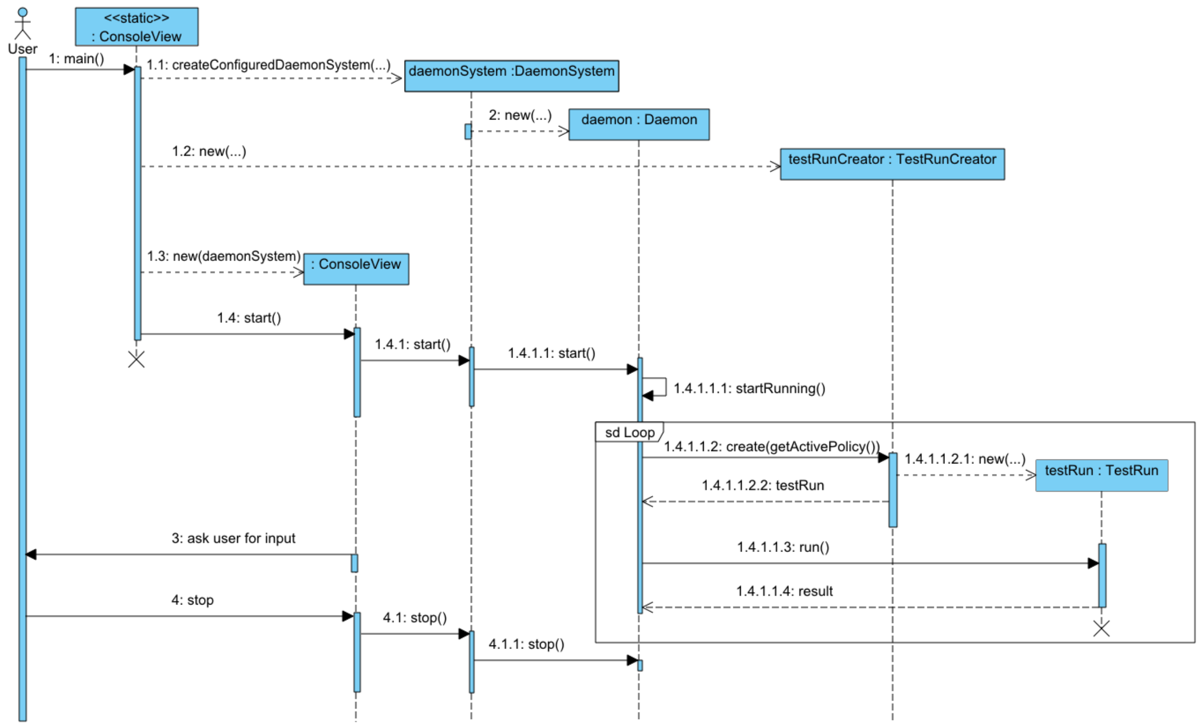


Figure 4: Sequence diagram of the execution

Responsibility pattern. As can be seen in figure 3, a `ConsoleMenuAction` exposes the `handle` method, which handles a command string entered by the user. This method uses the `canHandle` method to determine whether the current menu action is the selected one. If it is, the `execute` method is called, which concrete `ConsoleMenuActions` must implement. If it is not, the `handle` method of the successor `ConsoleMenuAction` is called, if there is a successor.

The `ChangePolicyMenuAction` is a menu action that sets a configured policy as the active policy on the `DaemonSystem`. Other menu actions are implemented in the `ConsoleView` by using anonymous classes implementing only the `execute` method. This is because the actions are highly specific, so that dedicated classes for these actions are not useful.

The `ConsoleMenu` class is responsible for showing a console menu and asking the user for choosing such a menu item. This class owns the first `ConsoleMenuAction` instance of the menu. It exposes a method to add other menu actions, which it delegates to the `addAction` method of that first `ConsoleMenuAction`. This `addAction` method in turn is implemented again as a Chain of Responsibility, to add the item to the end of the chain of `ConsoleMenuActions`. The `show` method of the `ConsoleMenu` prints all menu items to the console by using the `showMenuChain` method on its first `ConsoleMenuAction`. This `showMenuChain` method prints its own title and then calls the `showMenuChain` method on its successor menu item. After the menu has been printed, the user is asked to enter his choice.

The new classes that have been introduced simplify the `ConsoleView` significantly, since it now does not implement any menu handling behaviour anymore. The cohesion thus is significantly improved.

2.2.3 Moved classes

A smaller refactoring, in terms of written code and time effort, was to move certain classes to other or new packages.

Firstly, the package `testdatas`, which contains data items collected by `DataCollectors`, was made a subpackage in the `collectors` package. The same was done with the package `teststatistics`, which contains items summarized by `Statistics`: this package was made a subpackage in the `statistics`

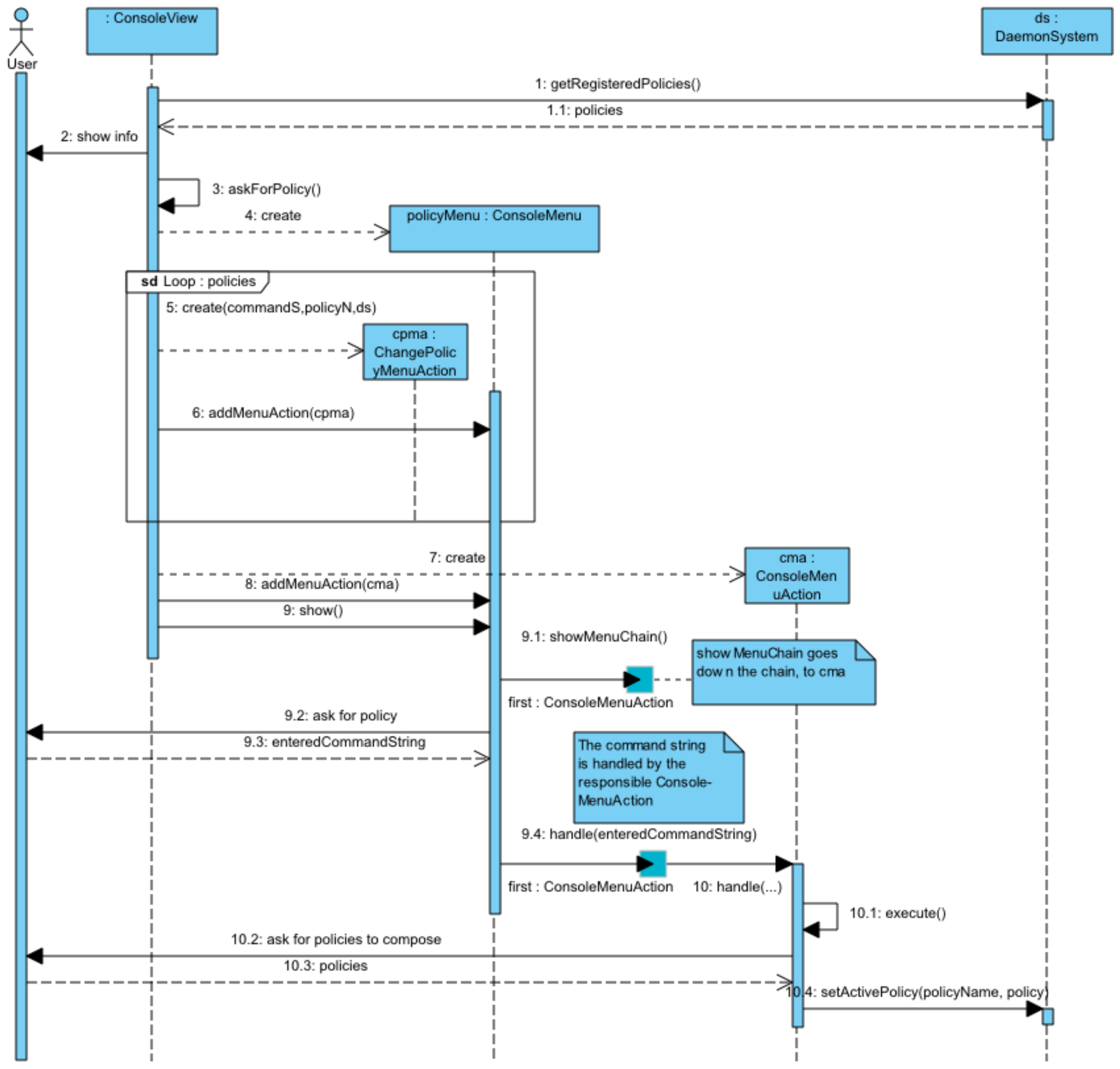

User
: ConsoleView
ds : DaemonSystem
1: getRegisteredPolicies()
1.1: policies
2: show info
3: askForPolicy()
4: create
policyMenu : ConsoleMenu
sd Loop : policies
5: create(commandS,policyN,ds)
cpma : ChangePolicyMenuAction
6: addMenuAction(cpma)
7: create
cma : ConsoleMenuAction
8: addMenuAction(cma)
9: show()
9.1: showMenuChain()
show MenuChain goes down the chain, to cma
first : ConsoleMenuAction
9.2: ask for policy
9.3: enteredCommandString
The command string is handled by the responsible Console-MenuAction
9.4: handle(enteredCommandString)
first : ConsoleMenuAction
10: handle(...)
10.1: execute()
10.2: ask for policies to compose
10.3: policies
10.4: setActivePolicy(policyName, policy)

Figure 5: Sequence diagram for interaction with the command line

package. These moves were done because these classes belonged more over there and they cannot be used without each other.

Secondly, as will be discussed in the analysis part, analysis tools reported cyclic dependencies between packages. This means that the modularity is bad: packages that have a cyclic dependency cannot be used independently, all of them are needed.

Cyclic dependencies occurred between our root package `kuleuven.group6` and some of its subpackages. The reason was that the root package uses most other packages, but some packages in turn also used the `FlattenedRequest` class and the `RunNotificationSubscriber` class. These classes were placed in the newly created `kuleuven.group6.testrun` package, together with the `TestRun` class and the `MethodRunner` class, that is used by the `FlattenedRequest`. This is done because all of these classes are related to a testrun. This restructuring removed the cyclic dependencies and allows to use each subpackage independently.

2.3 Analysis

We used several tools to analyse the refactored version of the project: InFusion, Understand and STAN. All of these tools generate software metrics to facilitate analysing the project.

2.3.1 InFusion

InFusion gives an overview of the complete software system by means of metrics. It can indicate design flaws and uses a quality deficit index as an indicator. These metrics can be used to identify flaws, but the numbers require further analysis to be of use.

We compare the initial *JUnit* project, the project including our daemon extension and our extension by itself. Figure 6 shows overview pyramids with the calculated metrics for these three cases. The figures show the following things:

- The cyclomatic complexity per line of code in the extension is higher than that of *JUnit* without the extension. This results in a higher value for the extended project, which now is an average value for Java. However, it still is a rather low value, which indicates that our code is not very complex.
- The tool indicates that the number of classes per package is rather small in both *JUnit* and in the extension, though slightly higher in the latter. This shouldn't be a problem as our packages are logical groupings of classes. If we were to provide more diverse functionality (for example, add more policies) this number would go up.
- The number of calls per method is good in the daemon extension. For *JUnit* with the extension it is rather small. Our code thus has more coupling dispersion, but it is an average coupling dispersion for Java.
- The NDD (Average number of direct descendants) and the HIT (Average height of the inheritance tree) are too high. The value of the NDD in our extension is half the value of that in *JUnit* separately. This means our class hierarchies are considerably less wide. Changes in superclasses thus don't have a great impact on the entire project, compared to *JUnit* . However, the value is still above average, which seems strange since our class hierarchies are not that wide at all. The metrics seem to prefer extremely shallow and narrow hierarchies, which seems contradictory to common OO principles. However, in light of the impact of changes, shallow and narrow hierarchies are better and less complex.

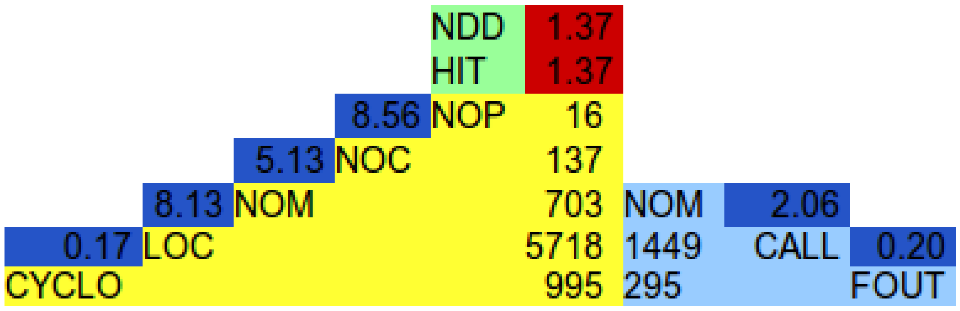
In general (*JUnit* with the extension) the classes are rather small, the methods are rather short and have an average logical complexity. They also call few other methods (low coupling intensity) from few other classes (low coupling dispersion). Although inFusion shows the NDD and HIT metrics in red (too high), we're not sure what the problem is, the hierarchies in the daemon project are not deeper or wider than they need to be. Comparison of the three overview pyramids shows that *JUnit* with the extension is quite similar to the original *JUnit* .

2.3.2 Understand

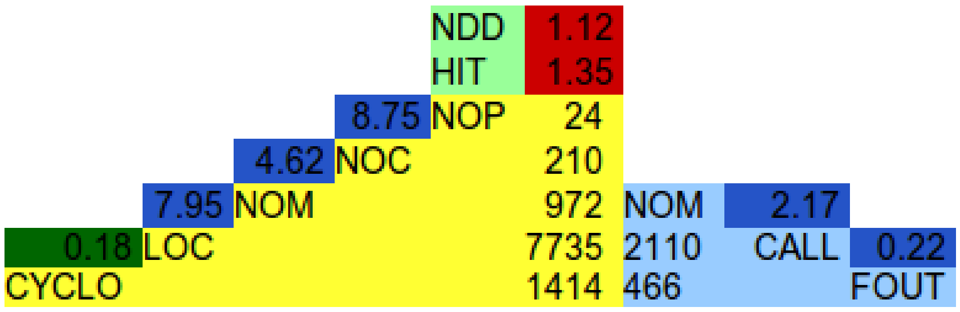
Another tool that we used to analyse our project was the tool Understand. Understand is a static analysis tool for maintaining, measuring and analyzing critical or large code bases. It can create dependency graphs, do a code check, can generate UML diagrams and more.

When we perform a codecheck on *JUnit* and the extension `Daemon` we have a result of 189 violations. 37 of them are from the daemon extension. The table with all the violations detected by Understand can be seen in table 2. The important violations of the unused instance variables are presented in red.

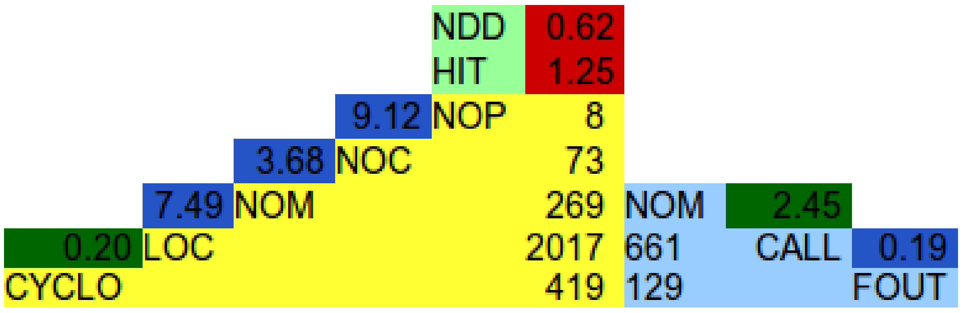
When we look at these violations we see that a lot of these violations are methods that are not called. Sometimes Understand can not detect that the methods are called, for example `Thread.run()` is not



(a) *JUnit* without the daemon extension



(b) *JUnit* with the daemon extension



(c) The daemon extension separately

Figure 6: Overview pyramid comparison

explicitly called, but is implicitly called by `Thread.start()`. 151 out of 189 are violations because a method is not called. These violations thus are useless.

22 violations are from variables that are not used. All of them except two come from the `static final long serialVersionUID` variable from the classes that are Serialized. These violations for the `serialVersionUID`, too, are of no importance. One of the other two violations is from an unused variable in `MethodRunner` in our extension and another is from a class in *JUnit* itself. This tells us that we sometimes should pay more attention.

16 violations are metric violations of the cyclomatic complexity. They tell us that the complexity of some of the methods is too high. 5 of these violations come from our extension: from the `CodeChangeCollector` and the `CompositeSortingPolicy`. In retrospect, these violating methods are too complex and should be simplified. This teaches us that it is easy to write complex code and care must always be taken to avoid this.

Most violations are not very important, but some violations show that there are still improvements to be made. It is therefore useful to use these kind of tools, since they point out problems that are easily overlooked in large projects.

Violation	*junit*	daemon extension	*org.junit*	*Total*
Unused instance variables	3 (0)	3 (1)	16 (1)	22 (2)
Unused methods	26	28	97	151
Cyclomatic Complexity	5	5	6	16
Total violations		37		189

Table 2: Table of violations from the tool *Understand*.
The numbers in red are the important ones.

2.3.3 STAN

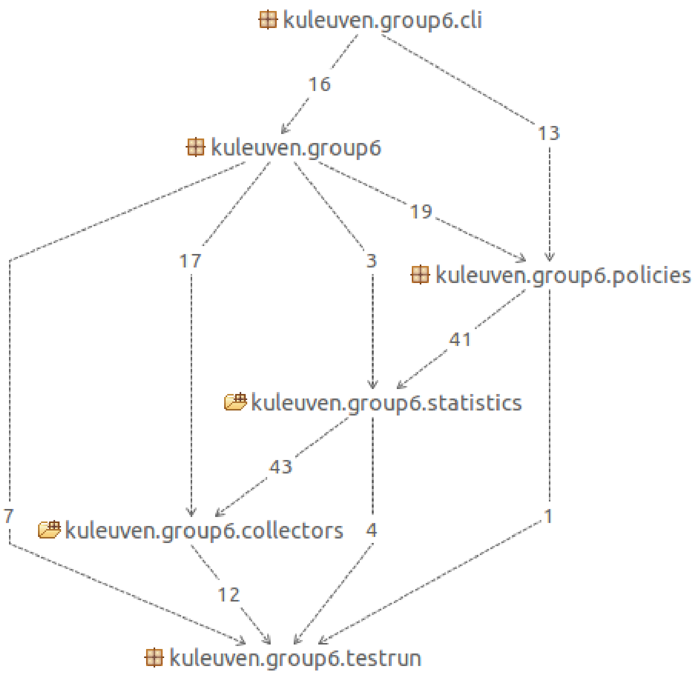


Figure 7: STAN Package Structure view

Structure Analysis for Java (STAN4J) is a code analysis tool that gives a good overview of the structure of a project. It calculates many similar metrics to inFusion.

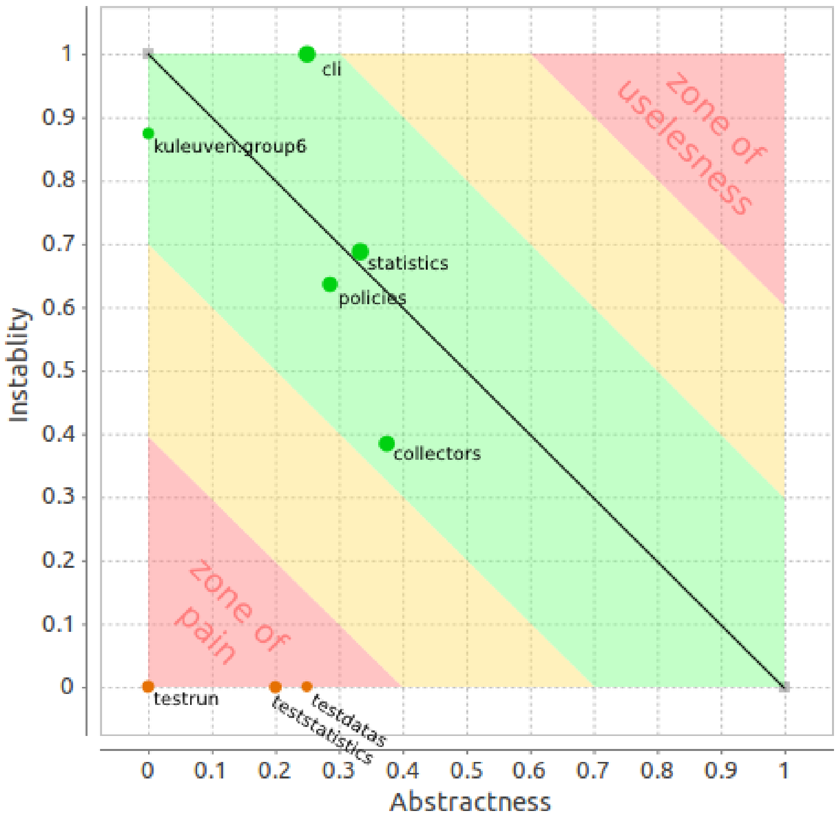


Figure 8: STAN Package Distance chart

STAN has a good dependency analysis view, which can be seen in figure 7. This allowed us to easily spot a couple of cyclic dependencies (STAN can show tangles of interdependent packages). Cyclic dependencies are to be avoided because they do not allow modular reuse of separate packages. These cyclic dependencies have been solved, as indicated in section 2.2.3.

Another informative feature of STAN is the Package Distance chart, seen in figure 8. Note that we added labels for the packages because figures do not have tooltips and we added colors which identify the regions that are "worse or better". This chart is based on the Stable Abstraction Principle: "The abstractness of a package should be in proportion to its stability." Most of our packages are reasonably close to the main sequence, three however are down to the far left in the "zone of pain".

Classes in the `testdatas` and `teststatistics` packages are very similar in functionality and implementation. They both contain one abstract class and several concrete implementations that do not depend on anything else. However, incoming dependencies are on these concrete implementations and not the abstraction. This is undesirable because concrete classes are more prone to change and cannot be as easily extended as abstract classes. `ITestData` and `ITestStatistic` are not really used as an abstraction (in normal use). This should be done differently, but it would change a lot of our design.

The third package in the "zone of pain" is the `testrun` package. This package contains classes that are all concrete and at the same time are heavily used in the project. The `TestRun` class could actually have an abstraction but we did not deem this necessary because this class has a very specific purpose. Making an abstraction for this class would improve the design, because other implementations could be made. The classes `FlattenedRequest` and `RunNotificationSubscriber` in the `testrun` package are also concrete classes that are heavily used by the rest of the project. We did not hide `RunNotificationSubscriber` behind an abstraction because it acts like a proxy for a stable class from *JUnit* (i.e. `RunNotifier`), the design would still be improved by adding this abstraction. `FlattenedRequest` extends `Request` and should be treated as a `Request` so it does not make sense to add a layer of abstraction between `Request` and `FlattenedRequest`.

Overall STAN helped us improve our package structure and highlighted a few points where we can still improve our project.

2.4 Conclusion

The implementation of the extension was relatively simple. We put a lot of effort in making our design extensible in the previous iteration.

The `Daemon` and the `ConsoleView` had to be refactored because they had too much responsibilities. The domain concept of a testrun was not clearly deliniated in the code in the previous design. We implemented two new classes that split the functionalities of `Daemon` . The initialization and configuration functionality from the `Daemon` were also extracted into the new `DaemonSystem` class. The new classes that have been introduced simplify the `ConsoleView` significantly. These refactorings significantly improve the cohesion. Along these more complex refactorings, other simpler refactorings were done: some classes were moved to other packages because they fit in better and because it improved modularity.

We inspected the project with multiple analysis tools. InFusion showed that *JUnit* with the extension is quite similar to the original *JUnit* . *Understand* warned of many violations but most of these were not relevant, although some of them show that there are still improvements to be made. The analysis with STAN made clear that the overal package structure was not optimal. We acted on this to improve our package structure but we can still further improve on our design.

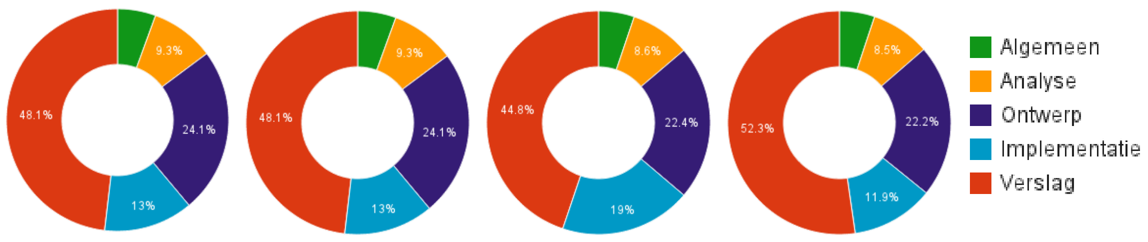
3 Project management

The division of tasks was about the same for everybody. Because it was a small task the design and the implementation were done together. In table 3 the workhours per teammember can be seen for the different parts of the assignment. The different parts are various (setting up eclipse, visual paradigm, *JUnit* , ..), design, implementation and report.

We had a lot discussion about the fairness of the ordening of the tests by a composite of policies. The assignment was not very clear and there were two different opinions about it. The implementation of the extension was not to difficult so it was easily done. We got a little stuck on the analyses because we did not know how to do it. Our first iteration was not very good so we were not sure how to improve the analyses for this iteration. In the end we found some usefull tools and we managed to do the analysis.

	Joren	Toon	Stef	Sophie
Various	01u30	01u30	1u30	01u30
Design	06u30	06u30	06u30	06u30
Implementation	03u30	03u30	05u30	03u30
Analyses	05u30	05u30	05u30	05u30
Report	08u00	14u30	14u30	14u30
Total	28u00	31u30	33u30	31u30

Table 3: Overview of the workhours per subject



(a) Joren (b) Toon (c) Stef (d) Sophie (e) Legend

Figure 9: Overview of the division of tasks

References

[1] *JUnit* consulted on 28/10/2013 via: http://junit.sourceforge.net/javadoc/

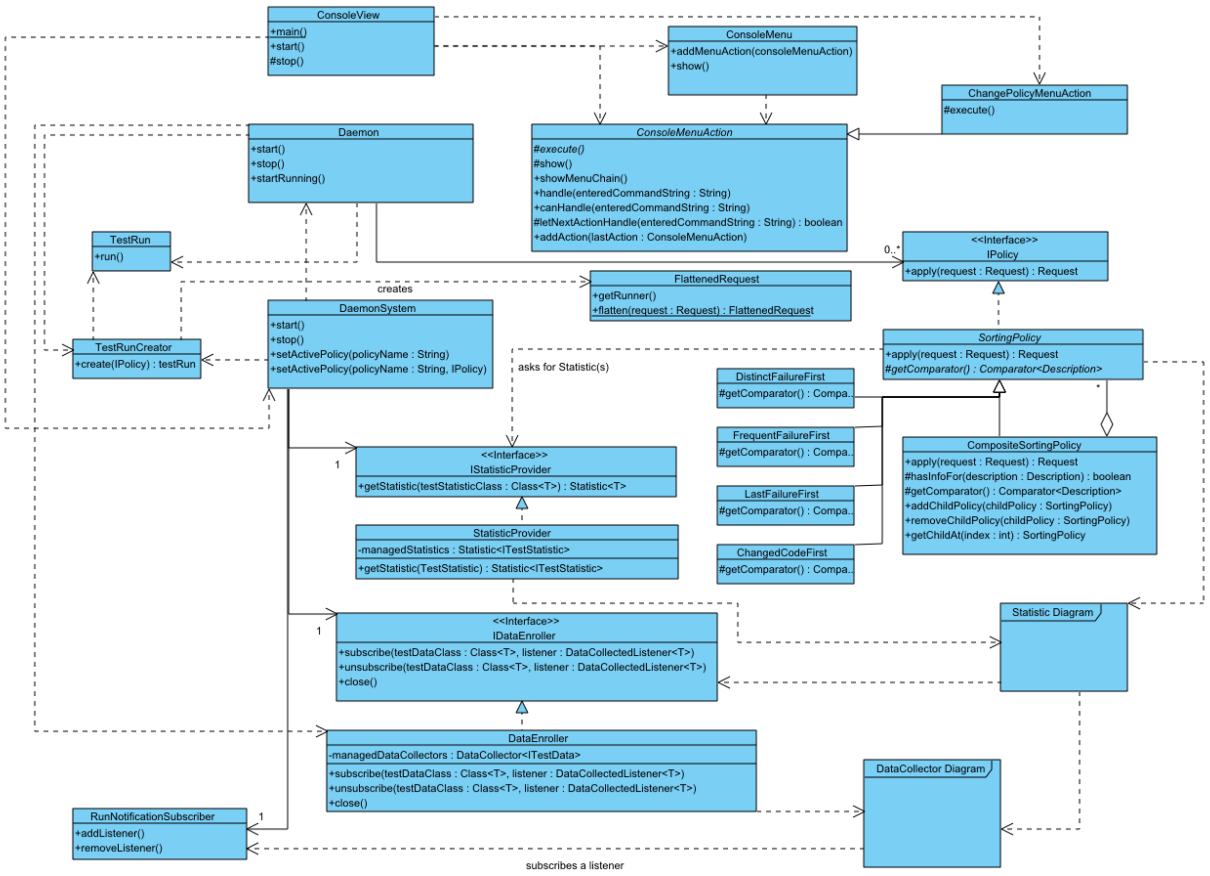
[2] *Applying UML and Patterns* consulted on 15/12/2013, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), by Craig Larman

[3] *InFusion Hydrogen* geraadpleegd op 19/12/2013 via: http://www.intooitus.com/products/infusion/download versie InFusion Hydrogen v1.8.0

[4] *Understand* geraadpleegd op 19/12/2013 via: http://emenda.eu/en/products/understand

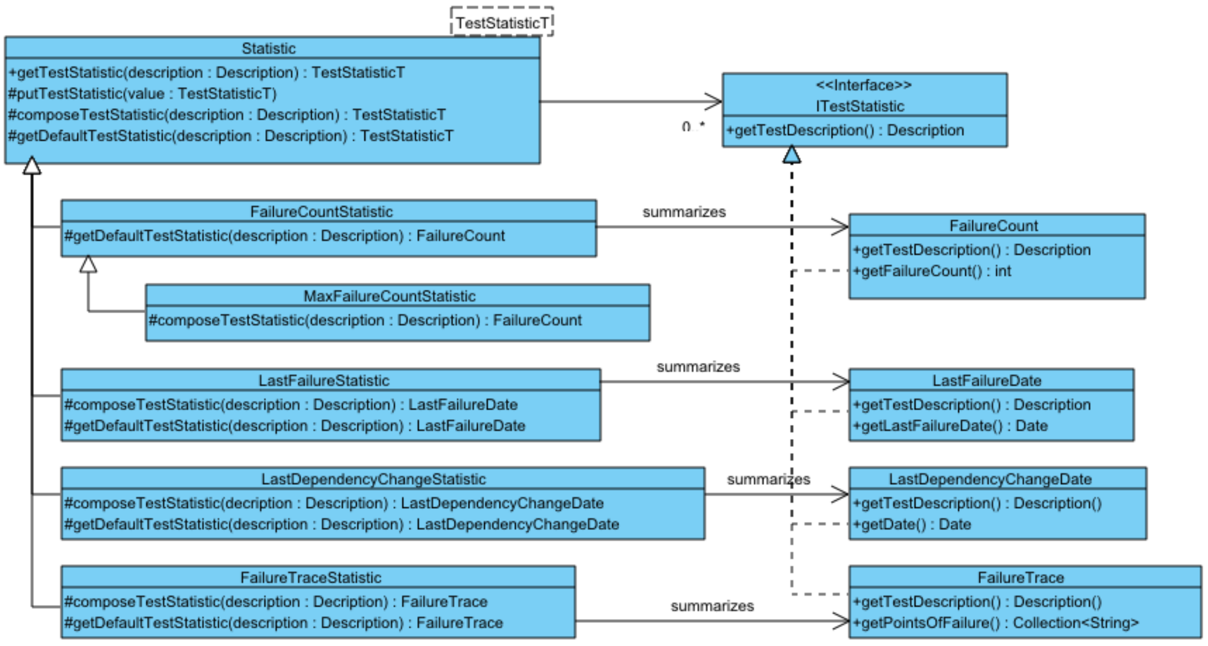
[5] *Stan4j* geraadpleegd op 19/12/2013 via: http://stan4j.com/ versie STAN 2.1

4 Appendix



ConsoleView
+main()
+start()
#stop()
ConsoleMenu
+addMenuAction(consoleMenuAction)
+show()
ChangePolicyMenuAction
#execute()
ConsoleMenuAction
#execute()
#show()
+showMenuChain()
+handle(enteredCommandString : String)
+canHandle(enteredCommandString : String)
#letNextActionHandle(enteredCommandString : String) : boolean
+addAction(lastAction : ConsoleMenuAction)
Daemon
+start()
+stop()
+startRunning()
TestRun
+run()
<<Interface>>
IPolicy
+apply(request : Request) : Request
0..*
creates
FlattenedRequest
+getRunner()
+flatten(request : Request) : FlattenedRequest
DaemonSystem
+start()
+stop()
+setActivePolicy(policyName : String)
+setActivePolicy(policyName : String, IPolicy)
TestRunCreator
+create(IPolicy) : testRun
SortingPolicy
+apply(request : Request) : Request
#getComparator() : Comparator<Description>
asks for Statistic(s)
DistinctFailureFirst
#getComparator() : Compa...
FrequentFailureFirst
#getComparator() : Compa...
LastFailureFirst
#getComparator() : Compa...
ChangedCodeFirst
#getComparator() : Compa...
CompositeSortingPolicy
+apply(request : Request) : Request
#hasInfoFor(description : Description) : boolean
#getComparator() : Comparator<Description>
+addChildPolicy(childPolicy : SortingPolicy)
+removeChildPolicy(childPolicy : SortingPolicy)
+getChildAt(index : int) : SortingPolicy
<<Interface>>
IStatisticProvider
+getStatistic(testStatisticClass : Class<T>) : Statistic<T>
1
StatisticProvider
-managedStatistics : Statistic<ITestStatistic>
+getStatistic(TestStatistic) : Statistic<ITestStatistic>
<<Interface>>
IDataEnroller
+subscribe(testDataClass : Class<T>, listener : DataCollectedListener<T>)
+unsubscribe(testDataClass : Class<T>, listener : DataCollectedListener<T>)
+close()
1
Statistic Diagram
DataEnroller
-managedDataCollectors : DataCollector<ITestData>
+subscribe(testDataClass : Class<T>, listener : DataCollectedListener<T>)
+unsubscribe(testDataClass : Class<T>, listener : DataCollectedListener<T>)
+close()
DataCollector Diagram
RunNotificationSubscriber
+addListener()
+removeListener()
1
subscribes a listener

Figure 10: The total class diagram after the refactoring. The `Statistics` and `DataCollectors` subdiagrams are the same as in the previous iteration and are shown in figures 11 and 12.



TestStatisticT
Statistic
+getTestStatistic(description : Description) : TestStatisticT
#putTestStatistic(value : TestStatisticT)
#composeTestStatistic(description : Description) : TestStatisticT
#getDefaultTestStatistic(description : Description) : TestStatisticT
0..*
<<Interface>>
ITestStatistic
+getTestDescription() : Description
FailureCountStatistic
#getDefaultTestStatistic(description : Description) : FailureCount
summarizes
FailureCount
+getTestDescription() : Description
+getFailureCount() : int
MaxFailureCountStatistic
#composeTestStatistic(description : Description) : FailureCount
LastFailureStatistic
#composeTestStatistic(description : Description) : LastFailureDate
#getDefaultTestStatistic(description : Description) : LastFailureDate
summarizes
LastFailureDate
+getTestDescription() : Description
+getLastFailureDate() : Date
LastDependencyChangeStatistic
#composeTestStatistic(decription : Description) : LastDependencyChangeDate
#getDefaultTestStatistic(description : Description) : LastDependencyChangeDate
summarizes
LastDependencyChangeDate
+getTestDescription() : Description()
+getDate() : Date
FailureTraceStatistic
#composeTestStatistic(description : Decription) : FailureTrace
#getDefaultTestStatistic(description : Description) : FailureTrace
summarizes
FailureTrace
+getTestDescription() : Description()
+getPointsOfFailure() : Collection<String>

Figure 11: The subdiagram of `Statistics`

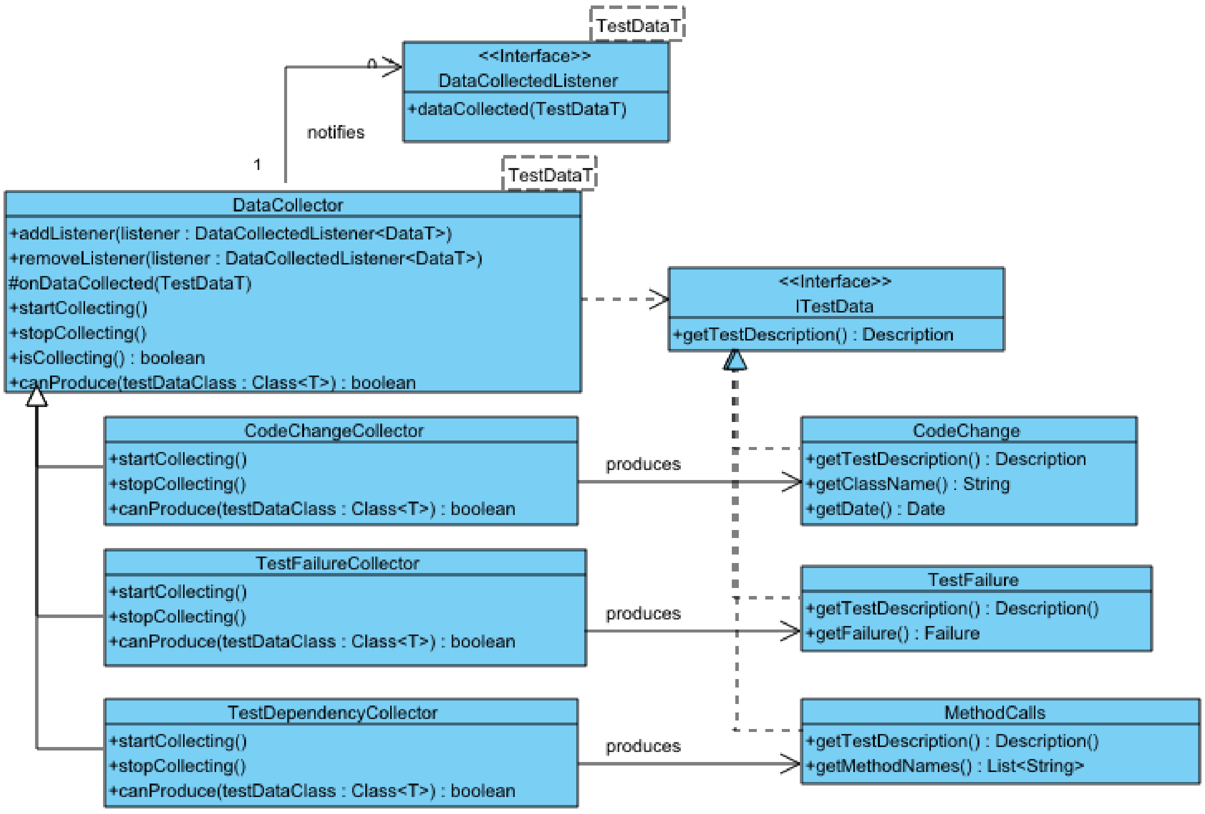

TestDataT
<<Interface>>
DataCollectedListener
+dataCollected(TestDataT)
0
notifies
1
TestDataT
DataCollector
+addListener(listener : DataCollectedListener<DataT>)
+removeListener(listener : DataCollectedListener<DataT>)
#onDataCollected(TestDataT)
+startCollecting()
+stopCollecting()
+isCollecting() : boolean
+canProduce(testDataClass : Class<T>) : boolean
<<Interface>>
ITestData
+getTestDescription() : Description
CodeChangeCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<T>) : boolean
produces
CodeChange
+getTestDescription() : Description
+getClassName() : String
+getDate() : Date
TestFailureCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<T>) : boolean
produces
TestFailure
+getTestDescription() : Description()
+getFailure() : Failure
TestDependencyCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<T>) : boolean
produces
MethodCalls
+getTestDescription() : Description()
+getMethodNames() : List<String>

Figure 12: The subdiagram of `DataCollectors`