KU LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN

Ontwerp van Softwaresystemen
Report: iteration 2

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

Computer Science 2013 – 2014

Contents

Introduction

Unit Tests are important because today's software systems are very complex. *JUnit* is the most popular framework for unit testing in the Java language. Currently, running the tests is either done manually by developers or automatically by a continuous integration tool on a central server. A disadvantage of these approaches is that tests only run when a developer decides to or when a developer commits work to the central repository.

This can be improved by using a daemon (background process) that continuously runs tests. This gives the developer rapid feedback and takes away the burden of manually starting tests.

The task of this iteration was to make a daemon that improves *JUnit* .

In section 1, the general decisions that are independent of the realisation in code are described. Section 2 describes how the system is designed. First an overview is given, after which the design is discussed in detail, together with class diagrams and sequence diagrams.

1 General decisions

A new testrun should be executed when source code changes, either in tests or in classes that are being tested. In hindsight this might be a problem because the output of a test might change even if the test or the code that is being tested doesn't change (for example if the test depends on some external resource that has time-dependent behavior like a network connection or a database). However, because it's better to use mock objects in those cases this was not considered in the current design.

A granularity of picked up changes has to be chosen. A distinction between code changes in different classes should obviously be made. It is however not immediately clear at which granularity changes inside a class should be detected. Changes in methods could be detected separately. Even a distinction between different execution paths inside a method could be made. However, we argue that a class should be coherent and as such, a change of the code inside a method would have a high possibility of impacting the behavior of the whole class. Therefore, code changes are looked at only at the level of classes and a finer distinction is not made.

Changes to the classes are detected by monitoring their class files. An alternative would be to monitor the actual Java source files, but most integrated development environments already offer automatic building functionality. Moreover, the test daemon would have to know the specifics of how to compile the entire project. Therefore, it is required that users of the test daemon use the automatic building feature of their IDE. The test daemon picks up changes in the compiled class files and as a result, it queues a new testrun.

Testruns should always be executed completely. Otherwise, when a developer is constantly making changes to the code, some tests would only be executed after a very long time. Only when the developer eventually takes a break longer than the duration of a testrun, these tests would be executed. While it could be argued that this is acceptable for some tests that are scheduled by a policy to run at the very end of a testrun, there are significant drawbacks to this approach.

An important drawback is starvation. Some policies can only change the execution order based on new information gained by running the tests. This would mean that for tests that are initially scheduled at the very end, it would be very hard to raise their priority, even though some code changes could significantly affect their test result. Even though techniques as aging could partially solve this starvation problem, the root of this problem would still be the same: new information on these tests is simply not available. Another solution would be to discard new information on all tests when the testrun has not completed entirely. However, this would undermine the purpose of the policy, since it could take very long until a policy takes new information into account.

Sometimes a comparison between tests that are run more often and tests that starve at the end of the testrun wouldn't even make sense. Consider the priority of tests under the frequent failure first policy.

It is not clear how to do a fair comparison between tests under this policy when some tests are run more often than others.

Because of these reasons, a testrun will always execute all tests.

2 Design

In this section we describe how our design followed from an analysis of responsibilities and we go through the start-up and execution of our application.

2.1 Design overview and responsibilities

The guiding principle for our design is a clear separation of responsibilities. The first step is an analysis of necessary actions:

- Run tests
- Collect information on tests
- Summarize the available information for a test
- Order tests according to a user-specified policy
- Generate output of results

Each of these is different enough to merit an entire entity responsible for that action. Those entities were implemented as `Daemon`, `DataCollector`, `Statistic`, `SortingPolicy` and `ConsoleView` respectively. Our daemon makes as much use of the existing *JUnit* infrastructure as possible.

The `Daemon` is responsible for executing testruns. It can make use of different policies, which influence the execution of testruns. For example, the `SortingPolicy` determines the order of the tests that are run.

An `IPolicy` is responsible for transforming a `Request` into a `Request` that is run according to the rules of the `IPolicy`.

`Statistics` summarize and store useful information about tests. This information can for example be used by sorting policies to define an ordering under the tests that are executed.

This information is a summary of data that is collected by `DataCollectors`. Examples of data that has to be collected for the supported policies are: every failure of a test, which code a test depends on and which code changes on disk during execution.

An `IPolicy` gains access to one or more `Statistics` through an `IStatisticProvider`. A concrete `IPolicy` sends the type of test statistic it needs to an `IStatisticProvider`, which then responds with a `Statistic` that can provide this type of test statistic. If it does not know about such a `Statistic`, an exception is thrown. This way, a layer of indirection is created between `IPolicys` and `Statistics`, so different implementations of a `Statistic` can easily be substituted by implementing `IStatisticProvider`.

`Statistics` use an `IDataEnroller` to subscribe to a certain kind of data. The actual data is collected by `DataCollectors`.

2.2 JUnit extensions

Our implementation does not modify anything in *JUnit* . The existing structure is used as much as possible. Only three classes were implemented which offer an extended or modified behavior of *JUnit* classes, as discussed below.

2.2.1 FlattenedRequest and MethodRunner

For this assignment, a great level of flexibility is needed for ordering tests in the best possible order, so that the most relevant tests are executed first and the most relevant results are presented as fast as possible to the user. Therefore, it should be possible to compare all the tests with each other.

The *JUnit* `Request` class is used to get a *JUnit* `Runner` which executes all tests. When a `Request` is created, all tests are configured in a hierarchy of `Runner`s. Consequently, the actual test methods are mostly nested a few levels deep in the hierarchy. It is not even required that all test methods are on the same level in the hierarchy. When a `Request` is sorted, for each node in the hierarchy, its direct children are sorted among each other. This is a problem, since an order should be imposed between *all* tests.

To accommodate this problem, the `FlattenedRequest` class is introduced. This class inherits from the *JUnit* `Request`, and it flattens all tests in an existing request, so that all test methods are located on the same level in the hierarchy. More specifically, the runner of a `FlattenedRequest` is a `Suite`, which directly contains test methods as its children (no suites). This class is mostly based on `MaxCore` in the *JUnit* experimental package. However, in `MaxCore`, the runners that represent a single test method are created via `Request.method`. This creates a runner for a test class, which is filtered for one test method.

This creates a problem: the *JUnit* `Description` for this runner is the `Description` of the test *class*. The runner therefore does not directly expose information on the test *method* that it actually executes. Since an order is imposed by looking at the `Description`s of tests, the goal of being able to sort all methods directly is not achieved.

To solve this problem, a custom runner is introduced: `MethodRunner`. The `Description` of this runner corresponds to the actual test *method*. A `MethodRunner` can be created for a certain test method in a certain class. From these details, the correct `Description` is created. Internally, this runner wraps another runner, which is created in the same way as in *JUnit* : via `Request.method`. However, the importance of the `MethodRunner` lies in the correct `Description`.

2.2.2 RunNotificationSubscriber

In *JUnit* a `RunNotifier` is created, which tests use to fire events during the execution of testunrs. Parties that are interested in these events can register a `RunListener` with this `RunNotifier`, so that they will be notified.

However, the `RunNotifier` also exposes methods to fire these events. It is therefore dangerous to just pass this object to all parties that want to subscribe themselves, since they could fire events, which we don't want them to do.

An alternative would be to ask for the listener of an interested party and register it ourselves. However, this is not our responsibility.

We chose to solve this problem by applying the Proxy pattern, more specifically, a protection proxy. In this pattern, access to an object is controlled. Here, this pattern is implemented by only exposing an `addListener` and `removeListener` method in the `RunNotificationSubscriber` class. These calls are delegated to the original `RunNotifier`. Other methods of the `RunNotifier` are not exposed. Whenever an object is interested in events of the testrun flow, it can be passed this `RunNotificationSubscriber`, by which it can subscribe itself.

2.3 Data collectors

`DataCollectors` are responsible for collecting necessary data during execution. Some `DataCollectors` collect data about tests that have been run, some collect data about changes in the testing environment such as files that have been changed. They link this information to a *JUnit* `Description`, which corresponds to a test method, test class or suite. When they have collected such a data item for a `Description`, they have to notify all interested parties of this event. Therefore, they also have to keep track of their interested parties.

2.3.1 Notification of interested parties

No assumptions should be made about who these interested parties are. The only thing a `DataCollector` should know, is that there is a party that is interested in the data it collects. This notification problem is a typical problem that can be solved by the Observer pattern. The `DataCollector` defines an interface by which interested parties can subscribe themselves: it has an `addListener` and a `removeListener` method, see figure 1. An interested party is a concrete class that realizes the `IDataCollectedListener` interface. This exposes the `dataCollected` method by which a `DataCollector` can notify the concrete listener. A `DataCollector` does not know the concrete type of the listener, and an `IDataCollectedListener` does not know the concrete type of the collector, as is prescribed by the Observer pattern. This minimizes the coupling between the two.

However, the Observer pattern has been implemented with a few modifications.

Firstly, when strictly applying the Observer pattern, a listener should check the new state of the collector when it is notified. However, the purpose of the collectors is not to store the data they collect, but only to pass this data to those who are interested. Collectors should therefore not hold state with respect to the collected data (they can, however, hold state with respect to the collection *process*). For this reason, the collector passes a collected data item directly to the notification method of the listener (the `dataCollected` method).

Secondly, there could be more than one way to collect the same data. However, an interested party does not need to know which one is being used. In fact, it doesn't even need to know that a `DataCollector` is being used to collect the data. It only cares about the data itself. For these reasons, subscribing and unsubscribing is done via an additional abstraction layer: via an `IDataEnroller`. There is another reason for introducing this abstraction layer, namely the creation and sharing of collectors, which is explained in more detail later in this section.

2.3.2 Types of collected data

Different data collectors collect different kinds of data. However, all kinds of collected data share one common property: they are linked to a *JUnit* `Description`, which corresponds to a test. To this end, the `ITestData` interface is defined, which specifies the `getTestDescription()` method. All concrete data collectors collect data of which the type is a subtype of `ITestData`. In general, an interested party will be subscribed for the specific concrete type of `ITestData` that the corresponding data collector collects. However, it could also be subscribed for the super type `ITestData`. Via the `getTestDescription()` method, it would know which test (or testclass or suite) corresponds with this data and it could pass this data to another object that knows better what to do with it. A simple example would be a logger that would log, per test, a string representation of each collected data item.

As can be seen in figure 1, the following three types of `ITestData` have been implemented:

TestFailure a failure of a test, produced by a `TestFailureCollector`

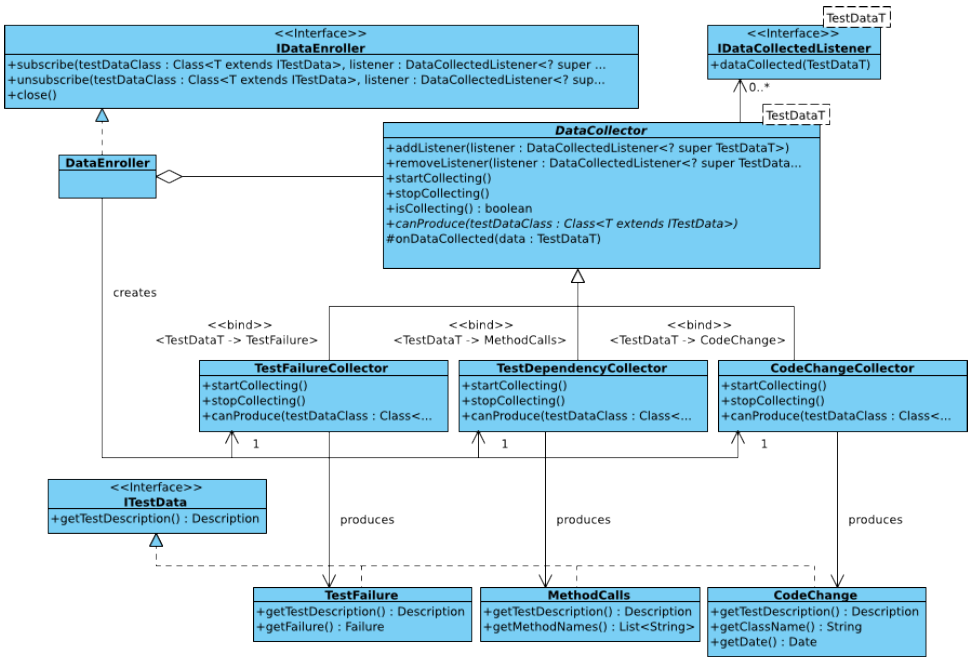
MethodCalls a list of method calls for a test, produced by a `TestDependencyCollector`

CodeChange a change in code that applies to a test, produced by a `CodeChangeCollector`

The verb *produce* is used here as opposed to collect, since a collector collects raw data and produces a user friendly version of it, rather than directly collecting existing instances of the `MethodCalls` class for example.

2.3.3 Data collectors for specific types of data

A concrete data collector is responsible for collecting only one kind of data. Even though different data collectors can collect different kinds of data, their interface remains the same, except for the type of the data itself. Moreover, an interested party is only interested in that kind of data, and nothing else.



<<Interface>>
IDataEnroller
+subscribe(testDataClass : Class<T extends ITestData>, listener : DataCollectedListener<? super ...
+unsubscribe(testDataClass : Class<T extends ITestData>, listener : DataCollectedListener<? sup...
+close()
TestDataT
<<Interface>>
IDataCollectedListener
+dataCollected(TestDataT)
0..*
TestDataT
DataCollector
+addListener(listener : DataCollectedListener<? super TestDataT>)
+removeListener(listener : DataCollectedListener<? super TestData...
+startCollecting()
+stopCollecting()
+isCollecting() : boolean
+canProduce(testDataClass : Class<T extends ITestData>)
#onDataCollected(data : TestDataT)
DataEnroller
creates
<<bind>> <TestDataT -> TestFailure>
<<bind>> <TestDataT -> MethodCalls>
<<bind>> <TestDataT -> CodeChange>
TestFailureCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<...
TestDependencyCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<...
CodeChangeCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<...
1
1
1
<<Interface>>
ITestData
+getTestDescription() : Description
produces
produces
produces
TestFailure
+getTestDescription() : Description
+getFailure() : Failure
MethodCalls
+getTestDescription() : Description
+getMethodNames() : List<String>
CodeChange
+getTestDescription() : Description
+getClassName() : String
+getDate() : Date

Figure 1: A class diagram of the package of data collectors

An interested party does not want to check that the data item which it is notified of, is actually of the type it is interested in. This could be forced by a contract of the subscription, i.e. that the listener will only be notified of collected data of the type it is interested in. However, the received data object would still have to be cast to the correct type to actually be able to access the collected data and do something with it.

For these reasons, the `DataCollector` class is generic in the type of data that it collects and notifies listeners about. Consequently, it is declared as `DataCollector<T extends ITestData>`. A concrete collector inherits from this class, filling in this generic type parameter `T`. This type must extend `ITestData`, since collectors collect a subtype of `ITestData`, as was elaborated on in the previous subsection.

Concrete collectors are not generic themselves. They have no reason to be, since they should know which data they collect. For example, a `TestFailureCollector<MethodCalls>` would not make sense. Therefore, the collectors are declared as follows, as is seen in figure 1:

TestFailureCollector extends DataCollector<TestFailure>
collects failures of tests

TestDependencyCollector extends DataCollector<MethodCalls>
collects the dependencies of a test, i.e. the methods that it calls

CodeChangeCollector extends DataCollector<CodeChange>
collects code changes that apply to tests

It follows that the `IDataCollectedListener` interface should also be generic (`IDataCollectedListener<T extends ITestData>`), since interested parties want to be notified only of a certain known concrete `ITestData` type. Because it is generic, a concrete listener can fix the type of data it is interested in at compile time, rather than at run time. Casts are not necessary and no additional checking of the type of passed data items is required.

2.3.4 Abstraction layer: IDataEnroller

As was noted earlier in section 2.3.1, the design would benefit from an abstraction layer around the data collectors.

The first reason was that an interested party does not need to know which implementation of `DataCollector` is used to notify it of a collected data item. It does not even have to know `DataCollector`s exist.

The second reason is that, even though their interface is simple, some data collectors may have an expensive mechanism, in terms of resource usage, to retrieve the data items. These collectors should be shared. For example, a `CodeChangeCollector` will have to watch directories and their subdirectories for file changes. If multiple parties are interested in code changes, they should use the same underlying `CodeChangeCollector` (whether they know it or not). The Observer pattern already allows the subscription of multiple listeners, but this does not mean that there is a good place where the listeners can access the collectors. Therefore, it is beneficial that an object exists which keeps track of all these collectors.

`IDataEnroller` is an interface that addresses this problem. Its only responsibility is to handle a subscription/unsubscription request for a listener for a specific kind of data. Through classes that implement this interface, parties that are interested in a certain type of `ITestData` can register and unregister a listener. As seen in the class diagram in figure 1, a listener has to communicate the class of the `ITestData` it is interested in.

This abstraction does not communicate anything about `DataCollector`s to its users, so the first reason is addressed. The second one is not explicitly solved: sharing of data collectors is not required by this abstraction, since it doesn't even have to use `DataCollector`s. However, the problem of sharing collectors is now moved to concrete `IDataEnroller`s. They can decide for themselves what is best.

In our design, the concrete class `DataEnroller` is an implementation of the `IDataEnroller` interface. This class will internally keep track of data collectors and will delegate a subscription or unsubscription request of listeners to a collector that supports the requested type of `ITestData`. It is however not its responsibility to know which collector this is. It is the collector itself that knows if it can produce a certain kind of `ITestData`, hence the method `canProduce(Class<T extends ITestData> testDataClass)` on the `DataCollector`. Internally, this `DataEnroller` will ask each of its tracked `DataCollector`s if it can produce the requested type of data, until it finds one that can. To this collector, it will delegate the subscription or unsubscription request.

If it does not find a collector that can produce the requested type of data, a `NoSuitableCollectorException` is thrown to indicate that the request can't be handled.

One thing that hasn't been discussed yet, is the creation of these collectors. It shouldn't be the responsibility of an interested party to create a collector. It is only interested in the data the collector produces. It also isn't the responsibility of the `DataEnroller`, since it doesn't have to know how to create each possible collector. However, due to limited time, this creational aspect is not entirely implemented as it should be. At this time, a static method `DataEnroller.createConfiguredDataEnroller()` is present that creates a new `DataEnroller`, together with the collectors that have been implemented. These created collectors are added to the new `DataEnroller` and the resulting instance is returned.

This creation, together with an example of a subscription of a listener, is illustrated in the sequence diagram in figure 2.

2.3.5 Improvement: chain of factories

Creating collectors is now done by `DataEnroller.createConfiguredDataEnroller()`. This is not an optimal solution, because this is not the responsibility of the `DataEnroller`. This is now implemented by putting the creational aspect inside a static method, that could easily be moved somewhere else. However, better designs exist.

Currently the `createConfiguredDataEnroller()` method must know how each collector can be created. It would be better to put the creation of each concrete `DataCollector` inside its own factory. The `DataEnroller` would then only need to create the factories and ask the relevant factory for a collector. The question remains how it can determine the relevant factory. A solution would be to organize all these factories in a chain of responsibility. Each factory would know for which concrete `ITestData` type it can create a collector. By doing this, the `DataEnroller` can ask the chain to create a collector for a

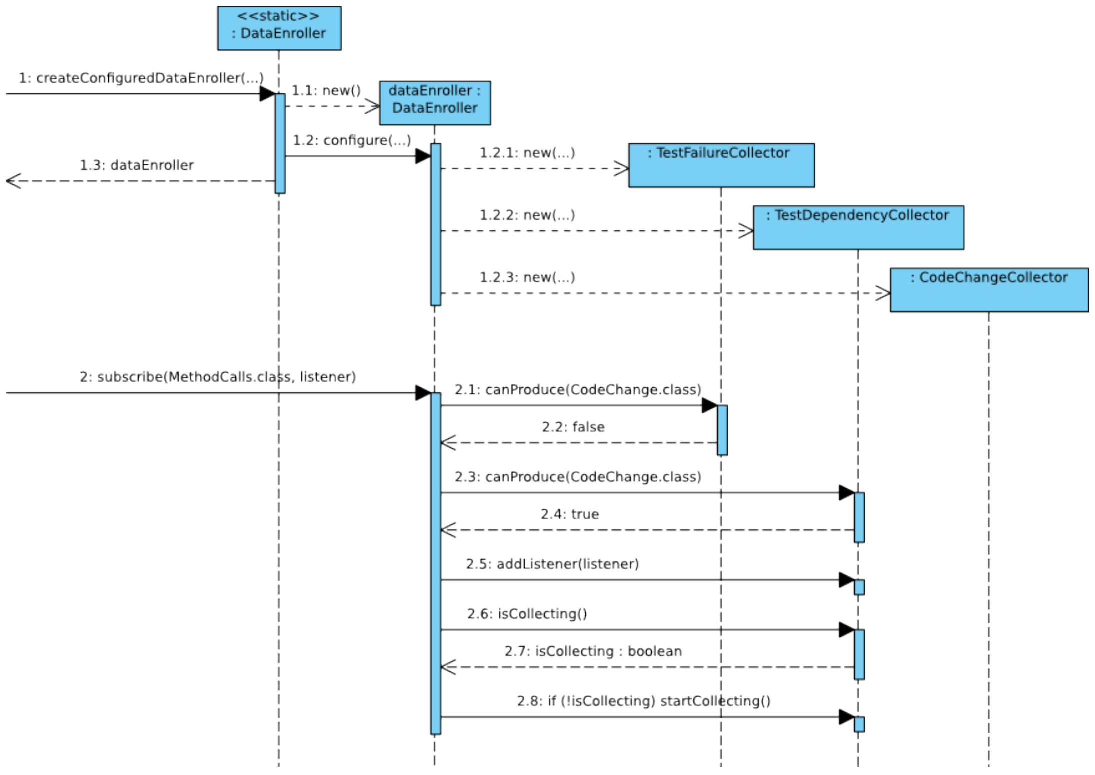

<<static>>
: DataEnroller
1: createConfiguredDataEnroller(...)
1.1: new()
dataEnroller : DataEnroller
1.2: configure(...)
1.3: dataEnroller
1.2.1: new(...)
: TestFailureCollector
1.2.2: new(...)
: TestDependencyCollector
1.2.3: new(...)
: CodeChangeCollector
2: subscribe(MethodCalls.class, listener)
2.1: canProduce(CodeChange.class)
2.2: false
2.3: canProduce(CodeChange.class)
2.4: true
2.5: addListener(listener)
2.6: isCollecting()
2.7: isCollecting : boolean
2.8: if (!isCollecting) startCollecting()

Figure 2: A sequence diagram of the subscription of a listener to DataEnroller

given class of `ITestData`. The first factory in the chain would check if it can handle the request. If not, it passes the request to the next factory. If no link in the chain can handle the request, an exception would again be thrown.

However, due to limited time, this improvement was not implemented.

2.3.6 Start and stop collecting

Some collectors have an expensive execution process. If they are not used, they should not execute. Furthermore, some collectors use resources that should be released when the program ends or when an isolated piece of code has reached its end. An example of these resources are threads. When a collector should only be used a limited amount of time, there should be a possibility to stop it. For this reason, a collector has the methods `startCollecting()` and `stopCollecting()`.

A simple situation that necessitates stopping collectors, is in unit tests. Different unit tests need to be executed in isolation. Therefore, it should be possible to stop the collectors.

2.3.7 Data collector implementations

Three types of collectors have been implemented, as can be seen in the class diagram in figure 1. They are all discussed in the following paragraphs.

2.3.7.1 TestFailureCollector

The `TestFailureCollector` extends `DataCollector<TestFailure>`. As such, it notifies interested parties of `TestFailures` it collects. When this collector is created, a `RunNotificationSubscriber` object is passed to it, on which it will subscribe an internal subclass that extends *JUnit*'s `RunListener`.

Subscribed `RunListeners` are notified by *JUnit* when events in the flow of a testrun occur. The event this collector is interested in, is the failure of a test. When this event occurs, it will wrap the *JUnit* failure inside a `TestFailure` instance, which this collector then passes to its subscribed, interested parties.

2.3.7.2 TestDependencyCollector

The `TestDependencyCollector` extends `DataCollector<MethodCalls>`. It will notify interested parties when it has collected a `MethodCalls` instance. This `MethodCalls` object contains a list of methods that have been called by a test. It is important that this is a list as opposed to another type of collection, since the order of executed methods may be important for some interested parties. The executed methods are detected by using the provided `OSSRewriter` package.

Only the code under test should be looked at by this collector. Therefore, this collector needs to know which classes belong to the code being tested. This is why the root directory of the code being tested is passed when creating this collector. It traverses this directory and keeps track of a list of all class files that it finds. At this time, the collector knows which classes should be looked at for method calls. Consequently, it sets an exclusion filter on the `OSSRewriter`, excluding other classes from being rewritten. The result is that notifications are only received for methods of the classes in the given directories.

The actual notification of called methods is done by subscribing a `Monitor` to the `MonitorEntrypoint`, of which the `enterMethod(String methodName)` is called when a method of a rewritten class is entered. Note: these classes are classes of the `OSSRewriter` package, we did not write them.

The only thing left to do, is determining which method call is executed by which test. This is why this collector also needs to be passed a `RunNotificationSubscriber` during creation. As with the `TestFailureCollector`, it will register an internal concrete `RunListener`, which listens to the *testStarted* and *testFinished* event. In response to the *testStarted* event, this collector creates a new list of called methods (represented by strings). The *testFinished* event notifies when a test has ended, so this is the time that the collected method calls are complete. Consequently, a `MethodCalls` object is created with the collected list of called methods and interested parties are notified. Of course, between the two events, each call the `enterMethod` method of the `Monitor` receives, adds the method it is being notified about to the list of method calls.

Unsolved issue Tests running in parallel cause problems with the approach discussed here, since every `Monitor` receives notifications for every method that is called. Thus, the `Monitor` can't easily make a distinction between methods executed by different tests.

One solution would be to check the stacktrace for each notification of a method execution. From this, the executing test could be determined. However, this approach would be very expensive, since a stacktrace would have to be retrieved every time the code being tested calls a method.

Another solution would be to check in which thread a called method is run. This is not always correct, since tests could cause other threads to be started.

Thus there is no clear solution for this problem. Fortunately, the parallel scheduling of tests is only an experimental feature in *JUnit* at this time, but in the future, this could become a problem.

OSSRewriter remark The `OSSRewriter` package has a static interface. Therefore, it is not easy to allow multiple parties to use this functionality. The static interface forces one party to be responsible for setting up the `OSSRewriter`. At this time, the `TestDependencyCollector` is the only class that uses it and only one instance is created of this collector. Due to limited time, that class was made responsible for enabling `OSSRewriter` and setting up the filter that excludes all code that is not code being tested. This should, however, be changed. Ideally, a wrapper should be made around the `OSSRewriter` package that is responsible for enabling it. This wrapper then could allow multiple parties to each register their own rewrite exclusion filter.

2.3.7.3 CodeChangeCollector

The `CodeChangeCollector` extends `DataCollector<CodeChange>`. It notifies its interested parties with `CodeChange` instances. A `CodeChange` contains the name of the class that has changed and the date that this change has occurred. The `CodeChangeCollector` needs to link a code change to a certain `Description`. However, it does not always know to which test this change corresponds, since it is not its responsibility to know which tests execute which code. Therefore, it links a code change to the *JUnit* `Description` of the root test suite class, since it is certain that the code change will apply to the root suite.

This collector uses a `WatchService` from the `java.nio.file` package. The test directory and the directory of code being tested is recursively registered with the `WatchService` for creation, deletion and modification events. This means that when polling the `WatchService` for changes, all new, deleted and modified files and directories are picked up, in all subdirectories of the test directory and the directory of code being tested.

When an event occurs for a file, we check that the file is a .class file. If it is, a code change is detected: a new `CodeChange` is created and all interested parties (subscribed listeners) are notified.

It is however also important that directories are watched. Each subdirectory in which the collector is interested, needs to be manually registered with the `WatchService`. Therefore, it is important that when a directory changes its name, when it is deleted or when a new directory is created, these changes are detected, in response to which the `WatchService` needs to be updated with the changes. Indeed, newly created directories or directories that have changed their name should be registered with the `WatchService`, otherwise these changes are completely ignored after such an event.

For example, Eclipse removes all subdirectories in the binary folder when a project is cleaned and rebuilt. If these events were not taken into account, changes would not be detected anymore after one clean and rebuild operation, which undermines the objectives of the test daemon.

Improvement As was said, a code change is notified of by linking it to the root testsuite `Description`. While it can be argued that one is certain that a code change indeed applies to the root suite, this information is not very useful. There are two alternatives.

The first would be that a data collector can also collect data items that are not tied to a *JUnit* `Description`. The responsibility of a data collector would then shift to the responsibility of collecting just *a* data item, instead of a data item that applies to a certain `Description`. However, a lot of classes notify others of new data, so this isn't really specific to a data collector.

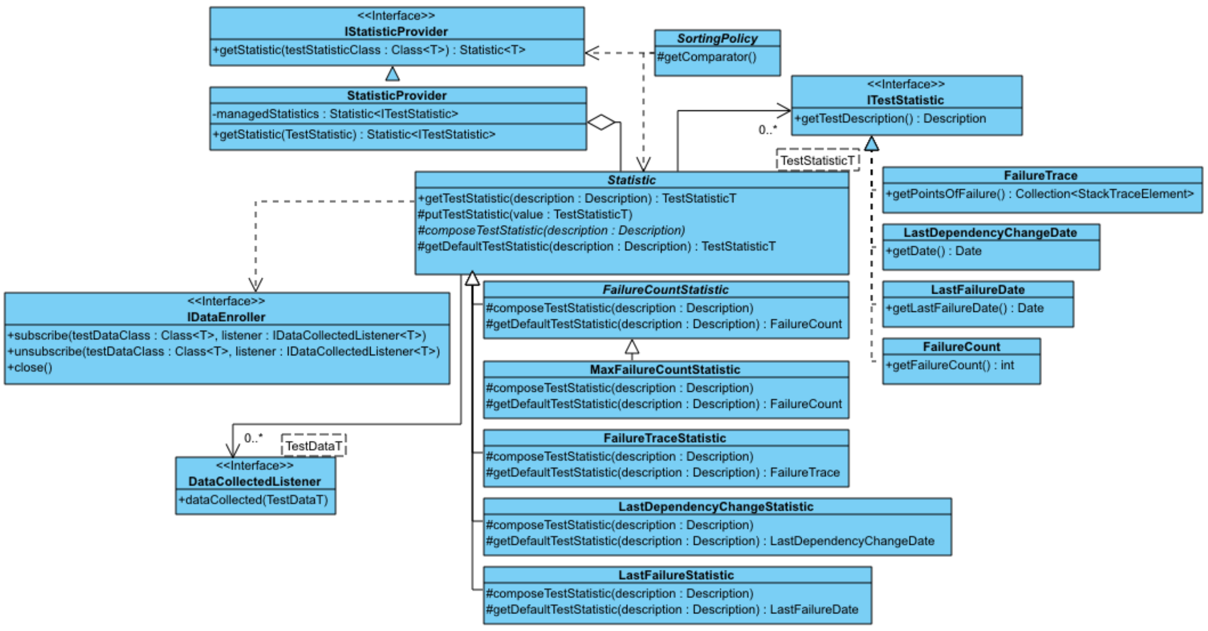
The other alternative would be that the `CodeChangeCollector` actually links the code change to the tests that are impacted by it. It would however not become responsible for computing this dependency information itself. Instead, it could retrieve this information from the `TestDependencyCollector`.

At this time however, none of these alternatives have been implemented.

2.4 Statistics

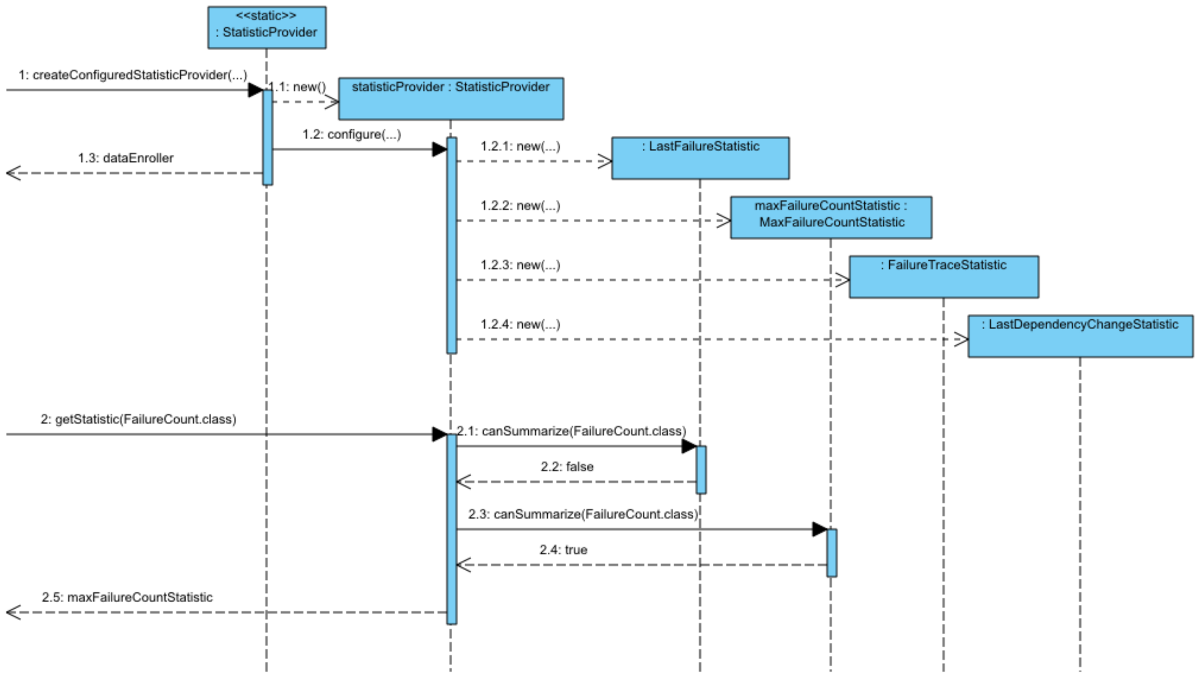
A `Statistic` is responsible for summarizing information about tests. As can be seen in figure 3, it has the method `getTestStatistic` to retrieve the information object for a specific `Description`. These information objects implement the `ITestStatistic` interface. Note the terms: a `Statistic` stores `ITestStatistics` that correspond to the test. The `ITestStatistic` interface specifies only the `getTestDescription()` method (perhaps not the most general name, since the `Description` it returns could be for a single test, a test class or a suite of tests). Each implementing class then provides methods by which more information can be retrieved. In figure 3, the different `ITestStatistic` realizations are shown, together with the information they add to the `ITestStatistic` itself.

As with the `DataCollectors` the abstract `Statistic` class is generic in the type of `ITestStatistic` it stores and returns. Concrete `Statistic` implementations fill in this generic parameter to indicate



<<Interface>>
IStatisticProvider
+getStatistic(testStatisticClass : Class<T>) : Statistic<T>
SortingPolicy
#getComparator()
StatisticProvider
-managedStatistics : Statistic<ITestStatistic>
+getStatistic(TestStatistic) : Statistic<ITestStatistic>
<<Interface>>
ITestStatistic
+getTestDescription() : Description
0..*
TestStatisticT
Statistic
+getTestStatistic(description : Description) : TestStatisticT
#putTestStatistic(value : TestStatisticT)
#composeTestStatistic(description : Description)
#getDefaultTestStatistic(description : Description) : TestStatisticT
FailureTrace
+getPointsOfFailure() : Collection<StackTraceElement>
LastDependencyChangeDate
+getDate() : Date
LastFailureDate
+getLastFailureDate() : Date
FailureCount
+getFailureCount() : int
<<Interface>>
IDataEnroller
+subscribe(testDataClass : Class<T>, listener : IDataCollectedListener<T>)
+unsubscribe(testDataClass : Class<T>, listener : IDataCollectedListener<T>)
+close()
FailureCountStatistic
#composeTestStatistic(description : Description)
#getDefaultTestStatistic(description : Description) : FailureCount
MaxFailureCountStatistic
#composeTestStatistic(description : Description)
#getDefaultTestStatistic(description : Description) : FailureCount
FailureTraceStatistic
#composeTestStatistic(description : Description)
#getDefaultTestStatistic(description : Description) : FailureTrace
0..*
TestDataT
<<Interface>>
DataCollectedListener
+dataCollected(TestDataT)
LastDependencyChangeStatistic
#composeTestStatistic(description : Description)
#getDefaultTestStatistic(description : Description) : LastDependencyChangeDate
LastFailureStatistic
#composeTestStatistic(description : Description)
#getDefaultTestStatistic(description : Description) : LastFailureDate

Figure 3: A class diagram of the package of statistics



<<static>>
: StatisticProvider
1: createConfiguredStatisticProvider(...)
1.1: new()
statisticProvider : StatisticProvider
1.2: configure(...)
1.3: dataEnroller
1.2.1: new(...)
: LastFailureStatistic
1.2.2: new(...)
maxFailureCountStatistic : MaxFailureCountStatistic
1.2.3: new(...)
: FailureTraceStatistic
1.2.4: new(...)
: LastDependencyChangeStatistic
2: getStatistic(FailureCount.class)
2.1: canSummarize(FailureCount.class)
2.2: false
2.3: canSummarize(FailureCount.class)
2.4: true
2.5: maxFailureCountStatistic

Figure 4: A sequence diagram of the retrieval of a statistic

which implementing type of ITestStatistic they return. The reasons are the same as with the DataCollectors. The generic Statistics eliminate the need of type checking that would be necessary in the Policys that request the collected concrete ITestStatistics.

A Statistic will typically summarize data it retrieves from DataCollectors. This mechanism is discussed later. However, a Statistic will not always have stored data for each and every test. For example, test failures are only collected on a test that actually fails, so its parent (the testclass) and possibly suites higher up in the hierarchy of tests have no failure that is collected for them.

In case of the Statistic a test statistic is requested for a Description for which no data is stored, the Statistic has the responsibility to check if the given Description is of an atomic test or a composed test (test class or suite or something else). If it is, it has to compose a new concrete ITestStatistic out of the test statistics it has of the children of the given Description. The Statistic retrieves the values for the child descriptions by calling the getTestStatistic method on itself, with the child description as its argument.

It is also possible that the given Description corresponds to an atomic test, and that no ITestStatistic is available for it. In this case, a default value must be constructed.

The getTestStatistic method does not know itself how to compose test statistics of the children of a Description. It also does not know which default values have to be used. These things are only known to the implementing concrete Statistic class. This can be solved by using the Template Method pattern. The getTestStatistic thus becomes a template method, which makes use of the abstract composeTestStatistic method and the abstract getDefaultTestStatistic method. These methods differ in behavior depending on the concrete type of ITestStatistic a subclass uses.

Concrete Statistics thus have to implement these abstract methods. The composeTestStatistic method can be used to calculate a compound statistic for a Description that is associated with a suite of tests. The concrete implementation of this method then uses the statistics for its children and produces some kind of summarized value from those (such as a sum, a maximum, an average, ...).

Following, the implemented Statistics are discussed in greater detail.

FailureCountStatistic collects the amount of times a test has failed until now. A test is identified by its Description. A FailureCount object is associated with this description, representing the failure count. When the DataCollector that this Statistic listens to signals that a test has failed, a new FailureCount is instantiated that is the incremented version of the previous one. This is done by the incremented() method of the FailureCount instance. The default value for the test statistic is of course failure count of zero. It is not clear how to compose different failure counts, there are multiple possibilities: the sum of failure counts of all the child Descriptions, the maximum of the failure counts of the children, etc. This is why the FailureCountStatistic is abstract and this choice is deferred to a subclass.

MaxFailureCountStatistic is a special kind of FailureCountStatistic. The only thing this class still has to implement, is the composeTestStatistic method. The choice here is to return the highest failure count of the failure counts of the child decriptions of the given Description.

LastFailureStatistic is a Statistic that stores the last time a test failed. This is done with a LastFailureDate object, of which the date of the last failure can be retrieved with the getDate() method. The composition of values of the child descriptions is easy: a new LastFailureDate should in that case be created of which the date equals the last date of the values for the child Descriptions.

FailureTraceStatistic is a Statistic<FailureTrace>. It is responsible for providing the points at which a certain test has failed. These points of failure are represented by the FailureTrace class. The actual points of failure are stored by a collection of strings that represent the methods in which a test has failed. An atomic test can of course only have one point of failure, but compound tests (testclasses, suites, etc.) can have more.

For each failing test in a test run a stack trace is retrieved by subscribing for a TestFailure with a IDataEnroller. This test failure contains information about the exception that caused the failure.

From this exception, the stacktrace can be retrieved. The deepest element in this stack trace is then the point of failure for this test.

This statistic, in contrast with other `Statistics`, does not keep its state across multiple test runs, since points of failure can not be summarized over multiple test runs. When a new test run starts, this statistic therefore clears all the stored test statistics (`FailureTraces`).

LastDependencyChangeStatistic is a `Statistic<LastDependencyChange>`, it is responsible for collecting the last change of a dependency of tests. This `Statistic` uses two different `ITestData` types to gather its statistics. Firstly, it subscribes to `MethodCalls` with a `IDataEnroller`, by which it knows which tests execute which methods of other classes. Secondly, is subscribes to `CodeChanges`, by which it knows of changes to classes in the project. This `Statistic` then combines these data into `LastDependencyChange` objects, by finding out which code changes apply to which test. These objects store the most recent date at which a dependency of the test it's associated with has changed.

`Policys` that need the statistic value for a certain test call the `getTestStatistic()` method on the `Statistic` object they received from an `IStatisticProvider`. The `Statistic` does not (need to) know about what kind of objects request this statistic data. The clients of the `Statistic` class also do not need to be notified when the statistic value for a test changes as they may only need a small subset of the available `ITestStatistics` and don't need to be up-to-date on all of them all of the time.

`Policys` and other parties that are interested in `Statistics` can retrieve a `Statistic` by using the `IStatisticProvider`. Concrete `IStatisticProviders` must implement the method `getStatistic`. It is responsible of giving back a `Statistic` that is capable of summarizing the specific type of `ITestStatistic` that is requested.

For example, an `IStatisticProvider` is passed to a `Policy` on creation. To get its needed `Statistic`(s), the policy passes the class of the concrete `ITestStatistic`(s) it requires to the `IStatisticProvider`, which then returns a `Statistic` that can produce this kind of `ITestStatistic`.

The class `StatisticProvider` implements such an `IStatisticProvider`. To know which `Statistic` object to return, the `StatisticProvider` keeps a `Set` of `Statistics` that are queried when the `getStatistic` method is called. The `StatisticProvider` calls the `canSummarize` method on every `Statistic` in the set with the given `ITestStatistic` class and returns the first `Statistic` for which this method returns *true*.

In this way, `Statistics` can be shared between `Policys`. All this information is therefore only stored once, but it also ensures consistency: if 2 different `Policys` request the same statistic for the same test, the response does not depend on the time that the `Statistic` that that `Policy` uses was created. It also makes switching `Policys` easier: when switching to another `Policy` that uses the same kind of `ITestStatistic`, statistics are already available that have the request data.

2.5 Policies

Policies determine certain properties of a testrun, examples are the order of tests and which tests are allowed to run (a filter). The design of the policies follows the Strategy pattern. Policies implement an interface `IPolicy` that specifies one method, `apply`, which transforms a `Request`. The abstract `SortingPolicy` defines the behavior of sorting policies with a template method, policies that extend this class have only to implement a method `getComparator` that returns a `Comparator` that can compare two `Description`'s. All implemented sorting policies make use of `Statistics` to determine an order for tests, but this is not required. At the moment, the following policies are supported: last failure first, frequent failure first, distinct failures first and changed code first. Each of these determines a different order that can be useful for developers. For example, tests that consistently order first under the frequent failure first policy might indicate over-complicated or fragile code.

Originally we wanted to implement policies so they could be combined, the composite pattern would be a good fit for this problem, we decided against this because it was not required by the assignment. However this would be a simple extension of the current design.

This combination of policies would be useful for example when you use the frequent failure first policy but many tests fail with the same frequency, you might want all tests with the same frequency ordered by the last failure first policy. Another use would be combining a policy that filters all tests that haven't failed yet with a sorting policy.

2.5.1 Frequent Failure First

The frequent failure first policy is used to identify tests that fail frequently. This would be useful because a particular test failing often, may indicate that the code it tests is too complicated, so developers often break it or bugs don't get fixed (because people introduce separate handling of exceptions when finding a bug takes too much effort). This policy makes use of `FailureCountStatistic`, tests with a higher number of failures go first, the order between tests with the same number of failures is undefined. (*JUnit* actually uses a stable sort, so this order would be the same as the original order, but this stability is not advertised in a contract and cannot be relied upon.)

2.5.2 Last Failure First

The last failure first policy prioritizes tests that have failed most recently. The assumption being that tests that recently failed will most likely fail again. This policy uses `LastFailureStatistic` which provides the `Date` of the last failure of a test. The order defined by this policy corresponds to the order of `Date`'s, most recent first.

2.5.3 Distinct Failures First

The distinct failures first policy specifies the most difficult ordering of tests. Even at the level of single tests (i.e. a single method in a testcase) the ordering is not trivial. This is because it doesn't really matter if a certain test goes before another or not, what matters is that distinct failures are tested as quickly as possible. For example if we have some tests (single methods) that failed in four different ways, let's say failures A, B, C and D, any test from one of these can go first, let's say a test from C goes first, then any test that failed on something different can go second, so now tests from A, B and D have priority over tests from C (A, B, D < C), when one test from every category has been executed, the order is again undefined.

This is a problem for the current implementation of *JUnit* 's `request` class. This class uses a comparator to sort all of its tests, but as we can see, you can not define an order between any two tests because that order depends on what tests have already been chosen. Because of this we decided that the distinct failures first policy would determine a complete order for all it's tests. This is a slight breach of the responsibilities of a sortingpolicy, a policy is responsible for determining an order for tests, however what tests it does this for should not be explicit in the policy.

When you need to sort not only single tests but entire suites of tests the ordering is even more difficult. One suite may cover a part of or everything another suite covers, one suite may cover everything in two or more other suites but contain more tests without a failure and thus be less interesting to execute even if the suites it covers partly overlap. This problem is similar to but not quite the same as the "Set covering problem" [4], which is np-hard. Because a complete and optimal solution is not critical and probably not worth the computational effort, we chose to implement a suboptimal solution. (i.e. a suite may go first if it contains a test that might have gone first if it was not in the suite. This suite could contain a method with a failure that has already been covered more times than any other but we don't check for this.) We also decided to order single tests before suites for a specific failure, because this increases the rate at which results for different failures can be shown, however between different failures a suite could go before a single test.

What follows is a description of the algorithm we implemented as a suboptimal solution to this problem. (This algorithm determines an order and we will mention the term bucket, however this algorithm has little to do with Bucket sort.) When the distinct failures first policy is applied it first determines a (nearly) total order for all the `description`'s in the request it is applied to. (It does not determine an order for `description`'s that have no failure.) The first step is to create a bucket for every failure that has happened during the previous testrun. Then the bucket's are filled with every test and suite that fail on a particular failure. A bucket contains every test(method) that fails on a certain failure and every suite that contains a test that fails on that failure and a counter for how many of its `description`'s have been selected. When the buckets have been created an iterative process of selection is started:

1. Choose a bucket with a minimal counter.
2. Pick a `description` from this bucket.
3. Remove this `description` from every bucket it appears in (in case of a suite).
4. Increment the counter for every bucket this `description` appeared in.
5. Record the `description` in a list that will contain a total order (only for failed tests, not for tests that did not fail) for all tests that failed.
6. Repeat until all buckets are empty (actually empty buckets are removed, so until you run out of buckets).

When using this policy's comparator the result is determined by the following rules: If both `description`'s appear in the list (total order) the order for the list applies. If only one `description` appears in the list, it is ordered first. If both `description`'s do not appear in the list then the order is not determined.

Remark In the original design, this policy was interpreted in a different way than it is implemented currently. Now, only points of failure are looked at. These are the methods that actually throw an exception. The previous approach was to look at each called method of a failing test, since test failures are not always caused by actual exceptions.

2.5.4 Changed Code First

The changed code first policy runs tests for which the containing class or the classes it tests have changed, first. This way new code or altered code is tested early on to give the developer feedback as soon as possible. This policy uses `LastDependencyChangeStatistic` which contains the `Date` of the last dependency change and sorts tests with the most recent dependency change date first.

2.6 Daemon

`Daemon` is 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. A data enroller and a statistic provider also is created.

Currently `Daemon` is implemented as a facade but at the moment it still contains too much functionality. For example testruns are now handled manually by `Daemon` , but this is not a responsibility of `Daemon` , a possible solution is to add a class that represents testruns.

Our implementation can be used without `Daemon` , but this requires knowledge of how to construct a `DataEnroller`, a `StatisticProvider`, a `RunNotifier`, etc.

`Daemon` is very similar to `JUnitCore`, however `JUnitCore` can be created and asked to run tests of multiple testclasses. `Daemon` however can only be asked to run a single top level suite class. This is not really a limitation, since it is easy to create a `Suite` class that specifies all the testclasses that need to be run and pass this `Suite` to `Daemon`.

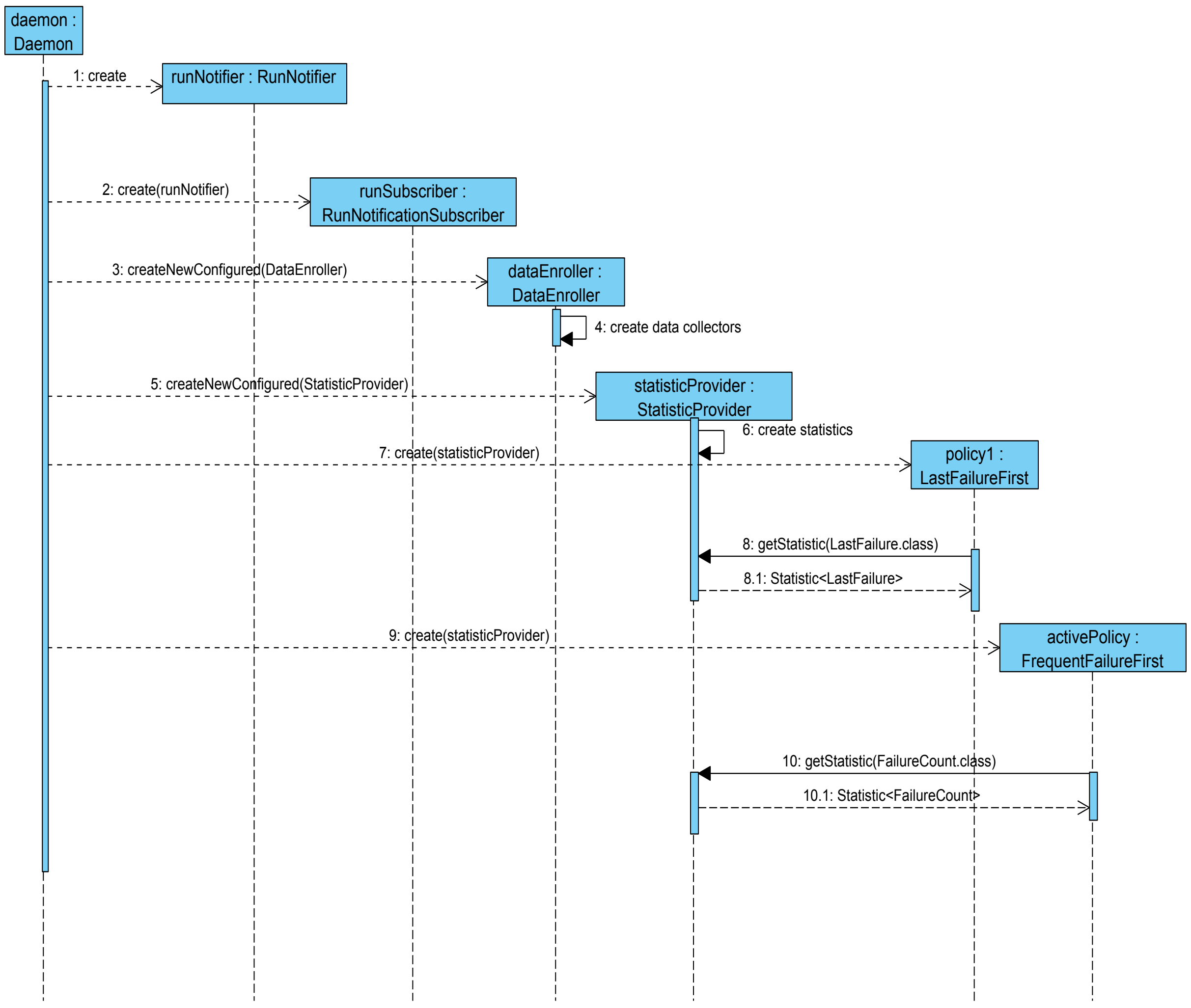

daemon : Daemon
1: create
runNotifier : RunNotifier
2: create(runNotifier)
runSubscriber : RunNotificationSubscriber
3: createNewConfigured(DataEnroller)
dataEnroller : DataEnroller
4: create data collectors
5: createNewConfigured(StatisticProvider)
statisticProvider : StatisticProvider
6: create statistics
7: create(statisticProvider)
policy1 : LastFailureFirst
8: getStatistic(LastFailure.class)
8.1: Statistic<LastFailure>
9: create(statisticProvider)
activePolicy : FrequentFailureFirst
10: getStatistic(FailureCount.class)
10.1: Statistic<FailureCount>

Figure 5: A sequence diagram of the initialization of `Daemon`

2.7 Interaction with the user

Users must be notified when new results of tests are available. Minimal interaction is required in the other direction: a user could possibly change the active policy and he could request to immediately run a new testrun, as opposed to waiting on code changes. The user also needs to be able to stop the program.

Different user interfaces should be supported. The first idea was to support this requirement by implementing the Model-View-Controller pattern. However, *JUnit* already has its own mechanisms to notify users of results through the use of `RunListeners`. It is beneficial to support this same mechanism, since a lot of editors already offer plugins to visualize *JUnit* results. With minimal effort, it should be possible to use our automatic test daemon instead of the standard `JUnitCore`. The `ConsoleView` adds a way to handle interaction with the user.

When the `ConsoleView` starts, the user is asked for the desired policy. It subscribes itself on the `Daemon` and starts it. For the events it receives, it outputs a textual representation.

Additionally, a menu is implemented: a user can always press the *<ENTER>* key, to view a list of options. Implemented options include changing the policy, queueing a new testrun and stopping the program.

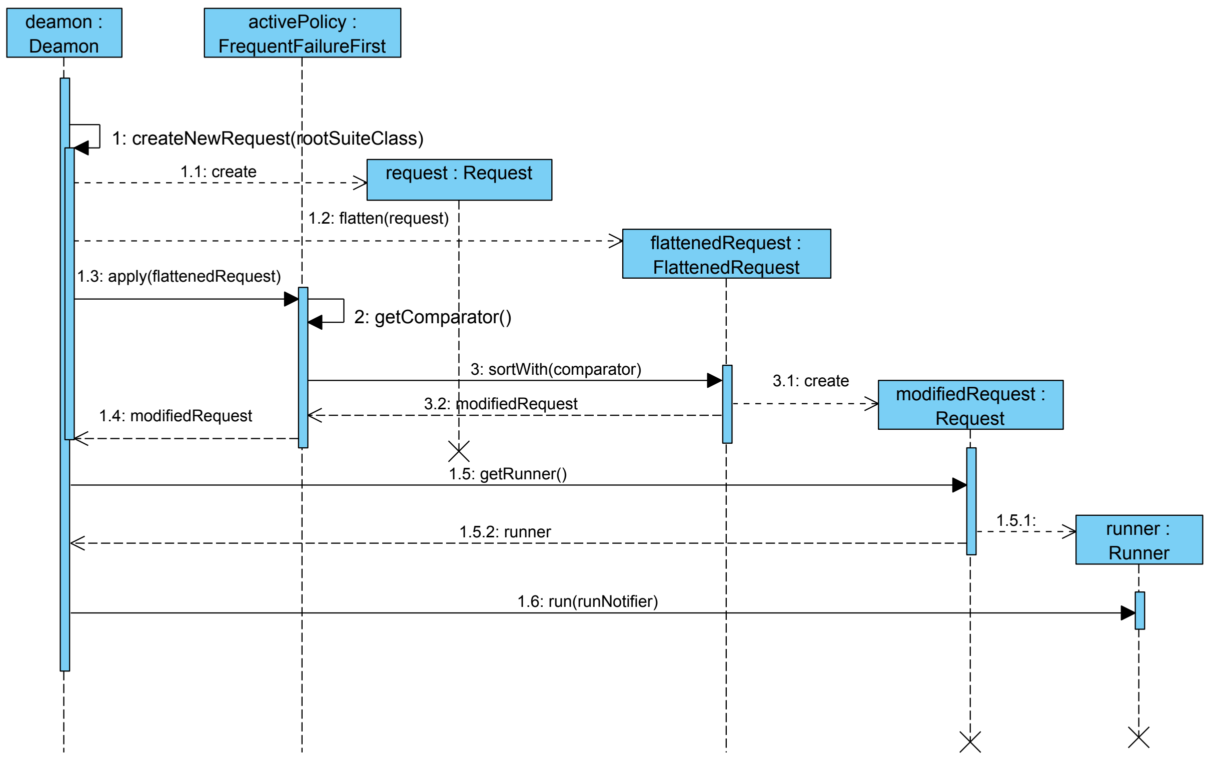


Figure 6: Sequence Diagram of the second part of the testrun

2.8 Execution of the program

The execution of the application starts in the `main` method. The `main` method needs three parameters. The first parameter is the top level suite of tests that need to be run. The second is the directory where the code is located that will be tested. The third parameter is the directory where the tests that will be executed are situated. If the parameters are not valid, an error message will be sent to the standard error stream. If they are valid, an instance of `Daemon` will be created, passing the three parameters. The `Daemon` will not start running tests at this time.

When a `Daemon` is created a `RunNotifier` is also created. Next, a `RunNotificationSubscriber` is created that wraps the `RunNotifier`. This object can be passed to entities that want to subscribe themselves on events during the testrun. The initialization of `Daemon` can be seen in figure 5.

Then, a `DataEnroller` is created. It needs to know of the `RunNotificationSubscriber`, the name of the top level suite class and the test and code directories, which have been passed to the `main` method. The `DataEnroller` will create multiple data collectors, which possibly need the former parameters. After that a `StatisticProvider` is created. This `StatisticProvider` will create `Statistics`. It needs to know of the `DataEnroller`, since statistics need to be able to subscribe themselves on collected data events. It also needs to know of the `RunNotificationSubscriber`.

After `Daemon` is created a new instance of `ConsoleView` is started. The `ConsoleView` will ask the user to provide an active policy out of the list of registered policies. When the user has chosen the active policy, the `ConsoleView` will start the `Daemon` .

`Daemon` will run a new testrun everytime code changes or a request for a new testrun is made.

When a new testrun is started, the first step is to create a new `URLClassLoader`, with the paths of the code directory and the test directory. By using this class loader, a new instance is created of the top level suite class, by specifying the name of that class which was provided. Next, a `Request` is created for the top level suite class. This process is shown in the sequence diagram in figure 6. First, a *JUnit* `Request` is created for the top level suite. This request then is flattened by the `FlattenedRequest` class, so that all method runners are located on the same level in the test hierarchy of the request. The next step is to apply the active policy to this flattened request. Different policies can do this in their own manner, but the implemented policies are sorting policies. Consequently, they ask the request to be sorted with a comparator they specify theirselves.

When the final request has been created, its runner is retrieved. Before that runner runs, the `fireTestRunStarted` method is invoked on the `RunNotifier`. This call indicates to the *RunNotifier* that the test run has started, it notifies all the registered `RunListener`s of this event. Following this, the runner is run. It is passed the `RunNotifier`, so that it can fire events during the testrun. Because the top level suite class was loaded by our class loader, all test classes are loaded by this same class loader. This ensures that the last version of each class is used and changes to the classes are picked up in a new testrun. When the testrun is finished, the `fireTestRunFinished` method is called to notify each `RunListener` that a testrun has finished.

3 Testing

For testing the application we made use of unit tests and a dummy project to use the application on. There's a unit test for each `DataCollector`, for the `LastFailureStatistic` and `MaxFailureCountStatistic`, and `FlattenedRequest`. These unit tests make use of stub objects to simulate the external components that some of these classes use. For example the test for `MaxFailureCountStatistic` uses a fake `DataEnroller` and `TestFailureCollector`. This dummy project also includes some stub classes which contain methods that exhibit behavior that is interesting to test on. For example there are classes that only include methods that throw exceptions. There are also tests in this project that fail non-deterministically with different rates of failure to enable us to check if the sorting according to a test's failure rate happens correctly. The `Policy`'s were tested by running the daemon on the dummy project (not an automatic test).

4 Project management

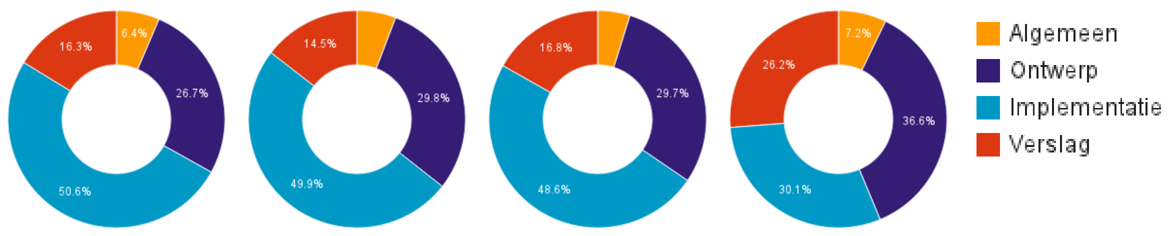
The division of tasks was about the same for everybody. We worked in group most of the time. In table 1 the workhours per teammember can be seen for the different parts of the assignment. The different parts are various (setting up eclipse, visual paradigm, *JUnit* , ..), design, implementation and report.

In the first weeks the emphasis was place on the design. Our classdiagram went through three iterations. We put a lot of time in making a good design. We did not always agree on everything and a lot of discussions followed. After a session with the assistant we always knew what still had to be improved in our design.

In the last half of week three we started implementing. One of the team members (Sophie) was on Athens that week and did not have time to help with the implementation. After the implementation was done, the whole group worked on the report. During the design and implementation notes were kept in the wiki on GitHub.

	Joren	Toon	Stef	Sophie
Various	3u30	4u00	4u00	4u00
Design	14u30	20u30	25u00	20u30
Implementation	28u00	34u30	40u30	16u50
Report	18u00	16u00	21u00	16u40
Total	64u00	75u00	90u30	58u00

Table 1: Overview of the workhours per subject



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

Figure 7: Overview of the division of tasks

5 Conclusion and discussion

The complete class diagram can be found in figure 8.

We believe our design has a few strong points. The existing *JUnit* code did not have to be modified. Only slight extensions of *JUnit* classes were needed.

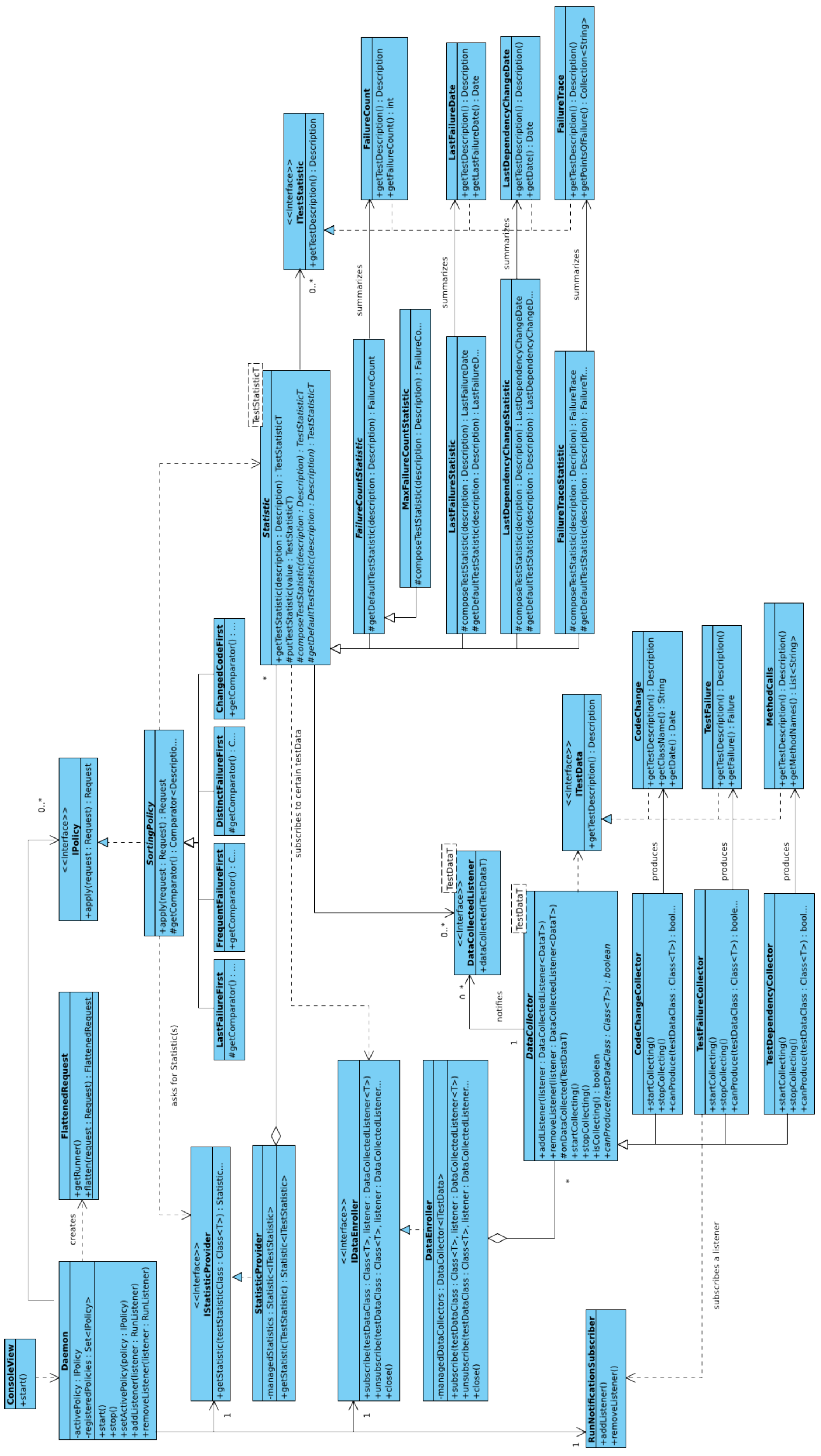
In general, the responsibilities in our design are clearly separated. Furthermore, it is easy to add new `DataCollectors`, `Statistics` or `IPolicys`.

In addition, because we used the *JUnit* result notification mechanism, it should be fairly easy to integrate our code with existing user interfaces, which are already available in a lot of IDE's.

Our design also has some weaker points. Because we started implementing at a late stage, we were not able to test the code as thoroughly as we should have. Additionally, some classes implement too much functionality. For example, `Daemon` now manages testruns at a very low level. `DistinctFailurePolicy` also contains an algorithm that is too complex. It would be better to encapsulate this algorithm in a separate class. It is also not ideal that the policy has to store the precalculated order of tests, as opposed to determining it when asked. Finally, the `CodeChangeCollector` does not fit in completely with the responsibilities of a `DataCollector`, since it doesn't actually link its data to a `Description`. However, these problems have been adressed in the report.

Possible improvements and extensions to the current design are:

- Add filtering policies.
- Allow policies to be combined.
- The responsibility for the creation of `DataCollectors` and `Statistics` can be improved with a Chain of Responsibility of factories.
- Allow statistics to be persisted.
- Integrate the system with existing GUI's.


ConsoleView
+start()
Daemon
-activePolicy : IPolicy
-registeredPolicies : Set<IPolicy>
+start()
+stop()
+setActivePolicy(policy : IPolicy)
+addListener(listener : RunListener)
+removeListener(listener : RunListener)
creates
FlattenedRequest
+getRunner()
+flatten(request : Request) : FlattenedRequest
<<Interface>>
IPolicy
+apply(request : Request) : Request
0..*
SortingPolicy
+apply(request : Request) : Request
#getComparator() : Comparator<Descriptio...
LastFailureFirst
#getComparator() : ...
FrequentFailureFirst
+getComparator() : C...
DistinctFailureFirst
#getComparator() : C...
ChangedCodeFirst
+getComparator() : ...
asks for Statistic(s)
<<Interface>>
IStatisticProvider
+getStatistic(testStatisticClass : Class<T>) : Statistic...
StatisticProvider
-managedStatistics : Statistic<ITestStatistic>
+getStatistic(TestStatistic) : Statistic<ITestStatistic>
1
<<Interface>>
IDataEnroller
+subscribe(testDataClass : Class<T>, listener : DataCollectedListener<T>)
+unsubscribe(testDataClass : Class<T>, listener : DataCollectedListener...
+close()
DataEnroller
-managedDataCollectors : DataCollector<ITestData>
+subscribe(testDataClass : Class<T>, listener : DataCollectedListener<T>)
+unsubscribe(testDataClass : Class<T>, listener : DataCollectedListener...
+close()
1
RunNotificationSubscriber
+addListener()
+removeListener()
1
subscribes a listener
subscribes to certain testData
*
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
MaxFailureCountStatistic
#composeTestStatistic(description : Description) : FailureCo...
LastFailureStatistic
#composeTestStatistic(description : Description) : LastFailureDate
#getDefaultTestStatistic(description : Description) : LastFailureD...
LastDependencyChangeStatistic
#composeTestStatistic(description : Description) : LastDependencyChangeDate
#getDefaultTestStatistic(description : Description) : LastDependencyChangeD...
FailureTraceStatistic
#composeTestStatistic(description : Description) : FailureTrace
#getDefaultTestStatistic(description : Description) : FailureTr...
summarizes
summarizes
summarizes
summarizes
FailureCount
+getTestDescription() : Description
+getFailureCount() : int
LastFailureDate
+getTestDescription() : Description
+getLastFailureDate() : Date
LastDependencyChangeDate
+getTestDescription() : Description()
+getDate() : Date
FailureTrace
+getTestDescription() : Description()
+getPointsOfFailure() : Collection<String>
TestDataT
<<Interface>>
DataCollectedListener
+dataCollected(TestDataT)
0..*
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
CodeChange
+getTestDescription() : Description
+getClassName() : String
+getDate() : Date
TestFailure
+getTestDescription() : Description()
+getFailure() : Failure
MethodCalls
+getTestDescription() : Description()
+getMethodNames() : List<String>
produces
produces
produces
CodeChangeCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<T>) : bool...
TestFailureCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<T>) : boole...
TestDependencyCollector
+startCollecting()
+stopCollecting()
+canProduce(testDataClass : Class<T>) : bool...
Figure 8: The complete classdiagram

6 Glossary

CodeChange
`CodeChange` contains the name of a class that has changed and the date of the change.

CodeChangeCollector
The class `CodeChangeCollector` collects data from files that have been modified in the directory of tests and in the directory of code being tested.

Collector
A collector collects information.

Comparator
A comparator imposes a total ordening on some collection of objects

ConsoleView
`ConsoleView` is the class that is the console view of the `Daemon` . It will give output to the user and the user can decide through the console view which policy has to be set as active policy.

DataCollector
A `DataCollector` is responsible for collecting a certain type of information on tests. Any type is possible because the class had a generic type.

DataEnroller
A `DataEnroller` class will internally keep track of data collectors and will delegate a subscription or unsubscription request of listeners to a collector that supports the requested type of `ITestData`.

Date
The class `Date` represents a specific instant in time, with millisecond precision.

Daemon
`Daemon` is responsible for executing the testruns. It can make use of different `Policy`'s to order the tests that need to be run.

Description
A `Description` describes a test which is to be run or has been run.

FailureCount
`FailureCount` is a teststatistic that contains the number of fails for a test

FailureCountStatistic
A `FailureCountStatistic` is a `Statistic` that will collects the number of times a certain test has failed.

FailureTraceStatistic
A `FailureTraceStatistic` is a `Statistic` that stores `FailureTrace` instances for each test it has data for. When a new testrun starts, all stored `FailureTrace` instances are removed, since they don't have a use anymore.

FlattenedRequest
A `FLattenedRequest` flattens the test hierarchy of a given of a given `Request` by putting all the leaves at the same, single level. This can be useful to allow ordening of test methods across the levels and nodes of the original hierarchy.

ITestData
The class `ITestData` contains the data for a test.

java.nio.file
java.nio.file determines a package that defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems.

JUnitCore
`JUnitCore` is a facade for running tests. It will run the testruns.

LastDependencyChange
LastDependencyChange is a teststatistic that contains the date of the last change to a dependency for a test.

LastDependencyChange is a Collector that collects dependencies for tests and one that collects which files have changed.

LastDependencyChangeStatistic
The LastDependencyChangeStatistic is a Statistic that makes use of two DataCollectors to create LastDependencyChange's. This Statistic will then combine that information to create LastDependecyChange's.

LastFailureDate
LastFailureDate is a teststatistic that contains the date of the most recent failure of a test.

LastFailureStatistic
The LastFailureStatistic is a Statistic that keeps the last time a test failed with a LastFailureDate object, which wraps a Date object.

MaxCore
MaxCore is a replacement for JUnitCore, which keeps track of runtime and failure history, and reorders tests to maximize the chances that a failing test occurs early in the test run.

MaxFailureCountStatistic
The class MaxFailureCountStatistic is a test class. It will check if the count is done right.

MethodCalls
The class MethodCalls has data of method calls for a test, that are produced by a TestDependencyCollector.

MethodRunner
A MethodRunner is a custom runner that has a description that directly corresponds with a testmethod.

Monitor
By subscribing a Monitor to the MonitorEntrypoint the notification of called methods is done. It receives notification for every method that is called.

OSSRewriter
The OSSRewriter can detect which code is being executed by a test. This tool has a java-agent that modifies the classes if they are being loaded in the virtual machine. At the beginning of each method, a callback is added to a monitor class. In this class self made monitors can be registered. In that way you can gather information on the execution of a test

Policy
A Policy imposes an order or filtering

Request
A Request is an abstract description of tests to be run.

rootSuiteClass
A rootSuiteClass will contain all the tests that will run if the Daemon is processed

RunListener
Junit provides support for adding listeners while executing the testcases via RunListener.

Runner
A **Runner** is created for each class implied by the **Request**. The Runner returns a detailed Description which is a tree structure of the tests to be run.

RunNotificationSubscriber
Whenever an object is interested in events of the testrun flow, it is passed to the RunNotificationSubscriber, by which it can subscribe to itself.

RunNotifier
A `RunNotifier` is a class, by which test fire events in the testrun flow. Parties that are interested in these events can register a `RunListener` with this `RunNotifier`, so that they will be notified.

Semaphore
A `Semaphore` maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used. So the `Semaphore` just keeps a count of the number available and acts accordingly.

SortingPolicy
A `SortingPolicy` determines an order for tests and suites. Examples of possible criteria for ordering: most frequent failures, shortest execution time.

Statistic
A `Statistic` listens to ITestData for tests and creates ITestStatistics for those tests.

StatisticProvider
A `StatisticProvider` keeps the statistics and it will provide the `Policy` with the right `Statistic`.

Testclass
A `TestClass` wraps a class to be run, providing method validation and annotation searching.

TestDependencyCollector
`TestDependencyCollector` is a `Collector` that collects all methods called by each test. This `Collector` assumes that the tests are run sequentially. If they are runned in parallel, the data collected by this `Collector` may be incorrect.

TestFailureCollector
A `TestFailureCollector` is a `Collector` that collects every failure for every test that is runned.

Testrun
The run that executes the tests to be run.

Unit Tests
`Unit Tests` are test that will test pieces of the sourcecode (units). For every unit there exist one or more tests.

WatchService
The `WatchService` is used from the `java.nio.file` package. It watches registered objects for changes and events.

References

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

[2] *OSSRewriter* consulted on 18/11/2013 via: http://toledo.kuleuven.be

[3] *Applying UML and Patterns* consulted on 1/11/2013, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), by Craig Larman

[4] *Set Covering Problem* consulted on 25/11/2013 via: http://en.wikipedia.org/wiki/Set_cover_problem