approval Documentation

Release 1.0

nikolavp

July 25, 2014

Contents

1 What can it be used for? 3
1.1 Getting Started . . . 3
1.2 User Manual . . . 7
1.3 Cool description of the library . . . 9
1.4 FAQ . . . 9
1.5 Javadoc . . . 9

Approval provides a powerful toolkit of ways to test the behavior of critical components so you can prevent problems **in your production environment.**

CHAPTER 1

What can it be used for?

Approval can be used for verifying objects that require more than a simple assert. The idea is that you sometimes just want to verify a particular result at the end and then start implementation refactoring. I like to call it "I will know the right result when I see it". Usecases for this might be:

- performance improvements to the implementation while preserving the current system output
- just verifying RESTful response results, be it JSON, XML, HTML whatever
- people use it for TDD but instead of providing the result upfront you just implement the simple possible thing, verify the result and then start improving the implementation.

1.1 Getting Started

***Getting Started* will guide you through the process of testing your classes with approval testing. Don't worry if you are used to normal testing with assertions you will get up to speed in minutes.**

1.1.1 Setting Up Maven

Just add the `approval` library as a dependency:

```
<dependencies>
    <dependency>
        <groupId>com.github.nikolavp</groupId>
        <artifactId>approval</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Note: Currently there are no stable version but we plan to release our first release soon, stay tuned :)

1.1.2 How is approval testing different

There are many sources from which you can learn about approval testing(just google it) but basically the process is the following:

1. you already have a working implementation of the thing you want to test
2. you run it and get the result the first time

3. the result will be shown to you in your preferred tool(this can be configured)
4. you either approve the result in which case it is recored(saved) and the test pass or you disapprove it in which case the test fails
5. the recorded result is then used on further test runs to make sure that there are no regressions in your code(i.e. you broke something and the result is not the same).
6. Of course sometimes you want to change the way something behaves so if the result is not the same we will prompt you with difference between the new result and the last recorded again in your preferred tool.

1.1.3 Approvals utility

This is the main starting point of the library. If you want to just approve a primitive object or arrays of primitive object then start here. The following will start the approval process for a `String` that `MyCoolThing` (our class under test) generated and use `src/test/resources/approval/string.verified` for recording/saving the results:

```
@Test
public void myApprovals() {
    String string = MyCoolThing.getComplexMultilineString();
    Approvals.verify(string, Paths.get('src/resources/approval/string.verified');
}
```

1.1.4 Approval class

This is the main object for starting the `approval` process. Basically it is used like this:

```
@Test
public void myApprovals() {
    String string = MyCoolThing.getComplexMultilineString();
    Approval<String> approver = Approval.of(String.class)
            .withReporter(Reporters.console())
            .build();
    approver.verify(string, Paths.get('src/resources/approval/string.verified');
}
```

note how this is different from *Approvals utility* - we are building a custom `Approval` object which allows us to control and change the whole approval process. Look at *Reporter class* and *Converter* for more info.

Note: Approval object are thread safe so you are allowed to declare them as static variables and reuse them in all your tests. In the example above if we have more testing methods we can only declare the `Approval` object once as a static variable in the *Test* class

---

1.1.5 Reporter class

Reporters(in lack of better name) are used to prompt the user for approving the result that was given to the `Approval` object. There is a `withReporter` method on `ApprovalBuilder` that allows you to use a custom reporter. We provide some ready to use reporters in the `Reporters` class:

- `console` - this uses **cat** and **diff** to report the first result or the differences on the console
- `gvim` - this uses **gvim** and **gvimdiff** to report the first result or the differences in gvim(our favourite editor)
- `gedit` - this uses **gedit** to report the first result. Sadly on differences it just opens two tabs :(

1.1.6 Converter

Converters are objects that are responsible for serializing objects to raw form(currently byte[]). This interface allows you to create a custom converter for your custom objects and reuse the approval process in the library. We have converters for all primitive types, String and their array variants. Of course providing a converter for your custom object is dead easy. Let's say you have a custom entity model class that you are going to use for verifications in your tests:

```
package com.nikolavp.approval;

/*
 * #%L
 * approval
 * %%
 * Copyright (C) 2014 Nikolavp
 * %%
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 * #L%
 */

public class Entity {

    private String name;
    private int age;

    public Entity(String name, int age) {
        this.age = age;
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

}
```

Here is a possible simple converter for the class:

```
package com.nikolavp.approval;

/*
 * #%L
 * approval
 * %%
 * Copyright (C) 2014 Nikolavp
```

```
 * %%
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 * #L%
 */

import com.nikolavp.approval.converters.Converter;

import javax.annotation.Nonnull;
import java.nio.charset.StandardCharsets;

public class EntityConverter implements Converter<Entity> {
    @Nonnull
    @Override
    public byte[] getRawForm(Entity value) {
        return ("Entity is:\n" +
                "age = " + value.getAge() + "\n" +
                "name = " + value.getName() + "\n").getBytes(StandardCharsets.UTF_8);
    }
}
```

now let's say we execute a simple test

```
    @Test
    public void customEntityTest() {
        Entity entity = new Entity("Nikola", 30);
        Approval<Entity> approver = Approval.of(Entity.class)
                .withReporter(Reporters.console())
                .withConveter(new EntityConverter())
                .build();
        approver.verify(entity, Paths.get("src/test/resources/approval/example/entity.verified"));
    }
```

we will get the following output in the console(because we are using the console reporter)

```
Entity is:
age = 30
name = Nikola
```

1.1.7 Path Mapper

Path mapper are used to abstract the way in which the final path file that contains the verification result is built. You are not required to use them but if you want to add structure to the your approval files you will at some point find the need for them. Let's see an example:

You have the following class containing two verifications:

now if you want to add another approval test you will need to write the same destination directory for the approval path again. You can of course write a private static method that does the mapping for you but we can do better with

PathMappers:

we abstracted the common parent directory with the help of the `ParentPathMapper` class. We provide other path mapper as part of the library that you can use:

- `JunitPathMapper`

1.1.8 Limitations

Some things that you have to keep in mind when using the library:

- unordered objects like *HashSet*, *HashMap* cannot be determisticly verified because their representation will vary from run to run.

1.2 User Manual

1.2.1 Simple example of the library

Let's try to test the simplest example possible:

```
package com.nikolavp.approval.example;

/**
 * User: nikolavp
 * Date: 19/03/14
 * Time: 19:16
 */
public class SimpleExample {
    public static String generateHtml(String pageTitle) {
        return String.format("<!DOCTYPE html>\n" +
                "<html lang=\"en\">\n" +
                "<head>\n" +
                "   <title>%s</title>\n" +
                "<meta charset=\"utf-8\"/>\n" +
                "<link href=\"css/myscript.css\"\n" +
                "      rel=\"stylesheet\"/>\n" +
                "<script src=\"scripts/myscript.js\">\n" +
                "</script>\n" +
                "</head>\n" +
                "<body>\n" +
                "...\n" +
                "</body>\n" +
                "</html>", pageTitle);
    }
}
```

now this class is not rocket science and if we want to test getResult(), we would something like the following in JUnit:

```
package com.nikolavp.approval.example;

import org.junit.Test;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;

/**
```

```
 * User: nikolavp
 * Date: 19/03/14
 * Time: 19:17
 */
public class SimpleExampleTest {

    @Test
    public void shouldReturnSomethingToTestOut() throws Exception {
        //arrange
        String title = "myTitle";
        String expected = "<!DOCTYPE html>\n" +
                "<html lang=\"en\">\n" +
                "<head>\n" +
                "   <title>" + title +"</title>\n" +
                "<meta charset=\"utf-8\"/>\n" +
                "<link href=\"css/myscript.css\"\n" +
                "      rel=\"stylesheet\"/>\n" +
                "<script src=\"scripts/myscript.js\">\n" +
                "</script>\n" +
                "</head>\n" +
                "<body>\n" +
                "...\n" +
                "</body>\n" +
                "</html>";

        //act
        String actual = SimpleExample.generateHtml(title);

        //assert
        assertThat(actual, equalTo(expected));
    }
}
```

this is quite terse and short. Can we do better? Actually approval is not any shorter:

```
package com.nikolavp.approval.example;

import com.nikolavp.approval.Approval;
import com.nikolavp.approval.reporters.Reporters;
import org.junit.Test;

import java.nio.file.Paths;

/**
 * User: nikolavp
 * Date: 19/03/14
 * Time: 18:11
 */
public class SimpleExampleApprovalTest {

    @Test
    public void shouldReturnSomethingToTestOut() throws Exception {
        //assign
        Approval<String> approver = Approval.of(String.class)
                .withReporter(Reporters.console())
                .build();
        String title = "myTitle";

        //act
```

```
        String actual = SimpleExample.generateHtml(title);

        //verify
        approver.verify(actual, Paths.get("./test.txt"));
    }
}
```

this is not the best example

1.3 Cool description of the library

1.4 FAQ

1.4.1 Illegal state exteption with "<myclass> is not a primitive type class!"?

This means that you are trying to create/use an `Approval` object that's for a non primitive type and you haven't specified a *Converter*

1.5 Javadoc

1.5.1 com.nikolavp.approval

Approval

public class **`Approval`**<T>

The main entry point class for each approval process. This is the main service class that is doing the hard work - it calls other classes for custom logic based on the object that is approved. Created by nikolavp on 1/29/14.

Parameters

- **<T>** – the type of the object that will be approved by this `Approval`

Constructors

Approval

`Approval` (Reporter *reporter*, Converter<T> *converter*)

Create a new object that will be able to approve "things" for you.

Parameters

- **reporter** – a reporter that will be notified as needed for approval events
- **converter** – a converter that will be responsible for converting the type for approval to raw form

Approval

`Approval` (Reporter *reporter*, Converter<T> *converter*, FileSystemUtils *fileSystemReadWriter*)

This ctor is for testing only.

Methods

getApprovalPath
public static Path **getApprovalPath** (Path *filePath*)
Get the path for approval from the original file path.

Parameters

- **filePath** – the original path to value

Returns the path for approval

getConverter
Converter<T> **getConverter** ()

getReporter
Reporter **getReporter** ()

of
public static <T> ApprovalBuilder<T> **of** (Class<T> *clazz*)
Create a new approval builder that will be able to approve objects from the specified class type.

Parameters

- **clazz** – the class object for the things you will be approving
- **<T>** – the type of the objects you will be approving

Returns an approval builder that will be able to construct an `Approval` for your objects

verify
public void **verify** (T *value*, Path *filePath*)
Verify the value that was passed in.

Parameters

- **value** – the value object to be approved
- **filePath** – the path where the value will be kept for further approval

Approval.ApprovalBuilder

public static final class **ApprovalBuilder**<T>
A builder class for approvals. This is used to conveniently build new approvals for a specific type with custom reporters, converters, etc.

Parameters

- **<T>** – the type that will be approved by the the resulting approval object

Methods

build
public Approval<T> **build** ()
Creates a new approval with configuration/options(reporters, converters, etc) that were set for this builder.

Returns a new approval for the specified type with custom configuration if any

withConveter
public ApprovalBuilder<T> **withConveter** (Converter<T> *converterToBeUsed*)
Set the converter that will be used when building new approvals with this builder.

Parameters

- **converterToBeUsed** – the converter that will be used from the approval that will be built

Returns the same builder for chaining

See also: `Converter`

withReporter
public ApprovalBuilder<T> **withReporter** (Reporter *reporterToBeUsed*)
Set the reporter that will be used when building new approvals with this builder.

Parameters

- **reporterToBeUsed** – the reporter that will be used from the approval that will be built

Returns the same builder for chaninig

See also: `Reporter`

Approvals

public final class **Approvals**
Approvals for primitive types. This is a convenient static utility class that is the first thing to try when you want to use the library. If you happen to be lucky and need to verify only primitive types or array of primitive types then we got you covered.

User: nikolavp (Nikola Petrov) Date: 07/04/14 Time: 11:38

Methods

verify
public static void **verify** (int[] *ints*, Path *path*)
An overload for verifying int arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **ints** – the int array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify** (byte[] *bytes*, Path *path*)
An overload for verifying byte arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **bytes** – the byte array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(short[] *shorts*, Path *path*)
An overload for verifying short arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **shorts** – the short array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(long[] *longs*, Path *path*)
An overload for verifying long arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **longs** – the long array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(float[] *floats*, Path *path*)
An overload for verifying float arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **floats** – the float array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(double[] *doubles*, Path *path*)
An overload for verifying double arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **doubles** – the double array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(boolean[] *booleans*, Path *path*)
An overload for verifying boolean arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **booleans** – the boolean array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(char[] *chars*, Path *path*)
An overload for verifying char arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **chars** – the char array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(String[] *strings*, Path *path*)
An overload for verifying string arrays. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **strings** – the string array that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(byte *value*, Path *path*)
An overload for verifying a single byte value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the byte that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(short *value*, Path *path*)
An overload for verifying a single short value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the short that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(int *value*, Path *path*)
An overload for verifying a single int value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the int that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(long *value*, Path *path*)
An overload for verifying a single long value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the long that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(float *value*, Path *path*)
An overload for verifying a single float value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the float that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(double *value*, Path *path*)
An overload for verifying a single double value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the double that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(boolean *value*, Path *path*)
An overload for verifying a single boolean value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the boolean that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify**(char *value*, Path *path*)
An overload for verifying a single char value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the char that needs to be verified
- **path** – the path in which to store the approval file

verify
public static void **verify** (String *value*, Path *path*)
An overload for verifying a single String value. This will call the approval object with proper reporter and use the path for verification.

Parameters

- **value** – the String that needs to be verified
- **path** – the path in which to store the approval file

DefaultFileSystemUtils

public class **DefaultFileSystemUtils** implements FileSystemUtils
A default implementation for `FileSystemUtils`. This one just delegates to methods in `Files`. User: nikolavp Date: 27/02/14 Time: 12:26

Methods

createDirectories
public void **createDirectories** (File *directory*)

move
public void **move** (Path *path*, Path *filePath*)

readFully
public byte[] **readFully** (Path *path*)

write
public void **write** (Path *path*, byte[] *value*)

FileSystemUtils

interface **FileSystemUtils**
This class is mostly used for indirection in the tests. We just don't like static utility classes. Created by ontotext on 2/2/14.

Methods

createDirectories
void **createDirectories** (File *parentPathDirectory*)

move
void **move** (Path *path*, Path *filePath*)

readFully
byte[] **readFully** (Path *path*)

write
void **write** (Path *path*, byte[] *value*)

Reporter

public interface **Reporter**
Created by nikolavp on 1/30/14.

Methods

approveNew
boolean **approveNew** (byte[] *value*, File *fileForApproval*, File *fileForVerification*)
Called by an `com.nikolavp.approval.Approval` object when a value for verification is produced but no old.

Parameters

- **value** – the new value that came from the verification
- **fileForApproval** – the approval file(this contains the value that was passed in)
- **fileForVerification** – the file for the this new approval value @return true if the new value is approved and false otherwise

Returns true if the value was approved and false otherwise

canApprove
boolean **canApprove** (File *fileForApproval*)
A method to check if this reporter is supported for the following file type or environment! Reporters are different for different platforms and file types and this in conjuction with `com.nikolavp.approval.reporters.Reporters.firstWorking` will allow you to plug different reporters for different environments(CI, Windows, Linux, MacOS, etc).

Parameters

- **fileForApproval** – the file that we want to approve

Returns true if we can approve the file and false otherwise

notTheSame
void **notTheSame** (byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)
Called by an `com.nikolavp.approval.Approval` object when values don't match in the approval process.

Parameters

- **oldValue** – the old value that was found in fileForVerification from old runs
- **newValue** – the new value that was passed for verification
- **fileForVerification** – the file for this approval value
- **fileForApproval** – the file for the new content

1.5.2 com.nikolavp.approval.converters

ArrayConverter

public class **ArrayConverter**<T> implements Converter<T[]>

An array converter that uses another converter for it's items. This allows this converter to be composed with another one and allow you to convert your types even if they are in an array. User: nikolavp Date: 20/03/14 Time: 19:34

Parameters

- **<T>** – The type of the items in the list that this converter accepts

Constructors

ArrayConverter

public **ArrayConverter**(Converter<T> *typeConverter*)

Creates an array converter that will use the other converter for it's items and just make array structure human readable.

Parameters

- **typeConverter** – the converters for the items in the array

Methods

getRawForm

public byte[] **getRawForm**(T[] *values*)

Converter

public interface **Converter**<T>

A converter interface. Converters are the objects in the approval system that convert your object to their raw form that can be written to the files. Note that the raw form is not always a string representation of the object. If for example your object is an image. User: nikolavp Date: 28/02/14 Time: 14:47

Parameters

- **<T>** – the type you are going to convert to raw form

Methods

getRawForm

byte[] **getRawForm**(T *value*)

Gets the raw representation of the type object. This representation will be written in the files you are going to then use in the approval process.

Parameters

- **value** – the object that you want to convert

Returns the raw representation of the object

Converters

public final class **Converters**

Converters for primitive types. Most of these just call toString on the passed object and then get the raw representation of the string result. . User: nikolavp Date: 28/02/14 Time: 17:25

Fields

BOOLEAN
public static final Converter<Boolean> **BOOLEAN**

A converter for the primitive or wrapper boolean object.

BOOLEAN_ARRAY
public static final Converter<boolean[]> **BOOLEAN_ARRAY**

A converter for the primitive boolean arrays.

BYTE
public static final Converter<Byte> **BYTE**

A converter for the primitive or wrapper byte types.

BYTE_ARRAY
public static final Converter<byte[]> **BYTE_ARRAY**

A converter for the primitive byte arrays.

CHAR
public static final Converter<Character> **CHAR**

A converter for the primitive or wrapper char object.

CHAR_ARRAY
public static final Converter<char[]> **CHAR_ARRAY**

A converter for the primitive char arrays.

DOUBLE
public static final Converter<Double> **DOUBLE**

A converter for the primitive or wrapper double object.

DOUBLE_ARRAY
public static final Converter<double[]> **DOUBLE_ARRAY**

A converter for the primitive double arrays.

FLOAT
public static final Converter<Float> **FLOAT**

A converter for the primitive or wrapper float object.

FLOAT_ARRAY
public static final Converter<float[]> **FLOAT_ARRAY**

A converter for the primitive float arrays.

INTEGER
public static final Converter<Integer> **INTEGER**
A converter for the primitive or wrapper int object.

INTEGER_ARRAY
public static final Converter<int[]> **INTEGER_ARRAY**
A converter for the primitive int arrays.

LONG
public static final Converter<Long> **LONG**
A converter for the primitive or wrapper long object.

LONG_ARRAY
public static final Converter<long[]> **LONG_ARRAY**
A converter for the primitive long arrays.

SHORT
public static final Converter<Short> **SHORT**
A converter for the primitive or wrapper short object.

SHORT_ARRAY
public static final Converter<short[]> **SHORT_ARRAY**
A converter for the primitive short arrays.

STRING
public static final Converter<String> **STRING**
A converter for the String object.

STRING_ARRAY
public static final Converter<String[]> **STRING_ARRAY**
A converter for an array of strings.

Methods

of
static <T> Converter<T> **of**()

ofArray
static Converter **ofArray**()

DefaultConverter

public class **DefaultConverter** implements Converter<byte[]>
Just a simple converter for byte array primitives. We might want to move this into `Converters`. User: nikolavp Date: 28/02/14 Time: 14:54

Methods

getRawForm
public byte[] **getRawForm**(byte[] *value*)

ListConverter

public class **ListConverter**<T> implements Converter<List<T>>
A list converter that uses another converter for it's items. This allows this converter to be composed with another one and allow you to convert your types even if they are in a list. User: nikolavp Date: 28/02/14 Time: 17:47

Parameters

- **<T>** – The type of the items in the list that this converter accepts

Constructors

ListConverter
public **ListConverter**(Converter<T> *typeConverter*)
Creates a list converter that will use the other converter for it's items and just make list structure human readable.

Parameters

- **typeConverter** – the converters for the items

Methods

getRawForm
public byte[] **getRawForm**(List<T> *values*)

ReflectiveBeanConverter

public class **ReflectiveBeanConverter**<T> implements Converter<T>
A converter that accepts a bean object and uses reflection to introspect the fields of the bean and builds a raw form of them. Note that the fields must have a human readable string representation for this converter to work properly. User: nikolavp Date: 28/02/14 Time: 15:12

Parameters

- **<T>** – the type of objects you want convert to it's raw form

Methods

getRawForm
public byte[] **getRawForm**(T *value*)

1.5.3 com.nikolavp.approval.reporters

ExecutableDifferenceReporter

public class **`ExecutableDifferenceReporter`** implements Reporter

A reporter that will shell out to an executable that is presented on the user's machine to verify the test output. Note that the approval command and the difference commands can be the same.

- approval command will be used for the first approval
- the difference command will be used when there is already a verified file but it is not the same as the value from the user

Constructors

ExecutableDifferenceReporter
public **`ExecutableDifferenceReporter`**(String *approvalCommand*, String *diffCommand*)

Main constructor for the executable reporter.

Parameters

- **approvalCommand** – the approval command
- **diffCommand** – the difference command

Methods

approveNew
public boolean **`approveNew`**(byte[] *value*, File *approvalDestination*, File *fileForVerification*)

canApprove
public boolean **`canApprove`**(File *fileForApproval*)

notTheSame
public void **`notTheSame`**(byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)

startProcess
Process **`startProcess`**(String... *cmdParts*)

FirstWorkingReporter

class **`FirstWorkingReporter`** implements Reporter

User: nikolavp (Nikola Petrov) Date: 14-7-21 Time: 15:35

Constructors

FirstWorkingReporter
`FirstWorkingReporter`(Reporter... *others*)

Methods

approveNew
public boolean **approveNew**(byte[] *value*, File *fileForApproval*, File *fileForVerification*)

canApprove
public boolean **canApprove**(File *fileForApproval*)

notTheSame
public void **notTheSame**(byte[] *oldValue*, File *fileForVerification*, byte[] *newValue*, File *fileForApproval*)

Reporters

public final class **Reporters**

Created with IntelliJ IDEA. User: nikolavp Date: 10/02/14 Time: 15:10 To change this template use File | Settings | File Templates.

Methods

console
public static Reporter **console**()

Creates a simple reporter that will print/report approvals to the console. This reporter will use convenient command line tools under the hood to only report the changes in finds. This is perfect for batch modes or when you run your build in a CI server

Returns a reporter that uses console unix tools under the hood

firstWorking
public static Reporter **firstWorking**(Reporter... *others*)

Get a reporter that will use the first working reporter as per `com.nikolavp.approval.Reporter.canApprove` for the reporting.

Parameters

- **others** – an array/list of reporters that will be used

Returns the newly created composite reporter

gedit
public static Reporter **gedit**()

Creates a reporter that uses gedit under the hood for approving.

Returns a reporter that uses gedit under the hood

gvim
public static Reporter **gvim**()

Creates a convenient gvim reporter. This one will use gvimdiff for difference detection and gvim for approving new files. The proper way to exit vim and not approve the new changes is with ":cq" - just have that in mind!

Returns a reporter that uses vim under the hood