Tested Objects 1.0 Users' Guide

FitNesse Integration for Naked Objects 4.0.x

Version 0.1

Preface

Tested Objects is a sister project for the Naked Objects framework, providing integration with FitNesse to enable agile acceptance (or scenario) testing. The integration bundles the FitNesse wiki server and provides a set of generic Fitnesse fixtures that interact with the domain model in the same manner that a Naked Objects viewer does. It also provides a Maven archetype to get you started quickly.

This user guide describes how to use Tested Objects to test your Naked Objects domain applications through FitNesse. Much of this guidance relates to using the Maven archetype; for ease of reference the archetype creates an online user guide within the FitNesse wiki. If you are interested in building Tested Objects from source code (perhaps with a view to contributing to or extending the capabilities of Tested Objects itself) then please see the developers' guide.

Tested Objects is hosted on SourceForge, and is licensed under Apache Software License v2. Naked Objects is also hosted on SourceForge, and is also licensed under Apache Software License v2.

Chapter 1

Introduction

1.1. Scenario Testing (aka Agile Acceptance Testing)

Prior to agile development, requirements gathering for systems was traditionally performed by business analysts discussing requirements with the business, and expressing those requirements in documentation, such as Word specs and perhaps spreadsheets. The acceptance criteria for such requirements were often only sketched out, if at all; it would normally fall to the system testers to write acceptance tests for the requirements, through a mixture of consulting the original (by now out-of-date) requirements documentation and (as often as not) reverse-engineering the implementation.

Scenario testing combines the requirements capture and the acceptance test criteria in a single form, through scenarios. As before, these requirements are in a form that a non-technical domain expert from the business can understand. What differs though is that these scenarios can be used directly exercise the system, and so also represent the acceptance tests for the scenario. Moreover, the results of these tests are rendered in such a way that the business can understand, and thus can help determine if the code is at fault or the test. Once implemented, the acceptance tests also act as a regression suite for the system.

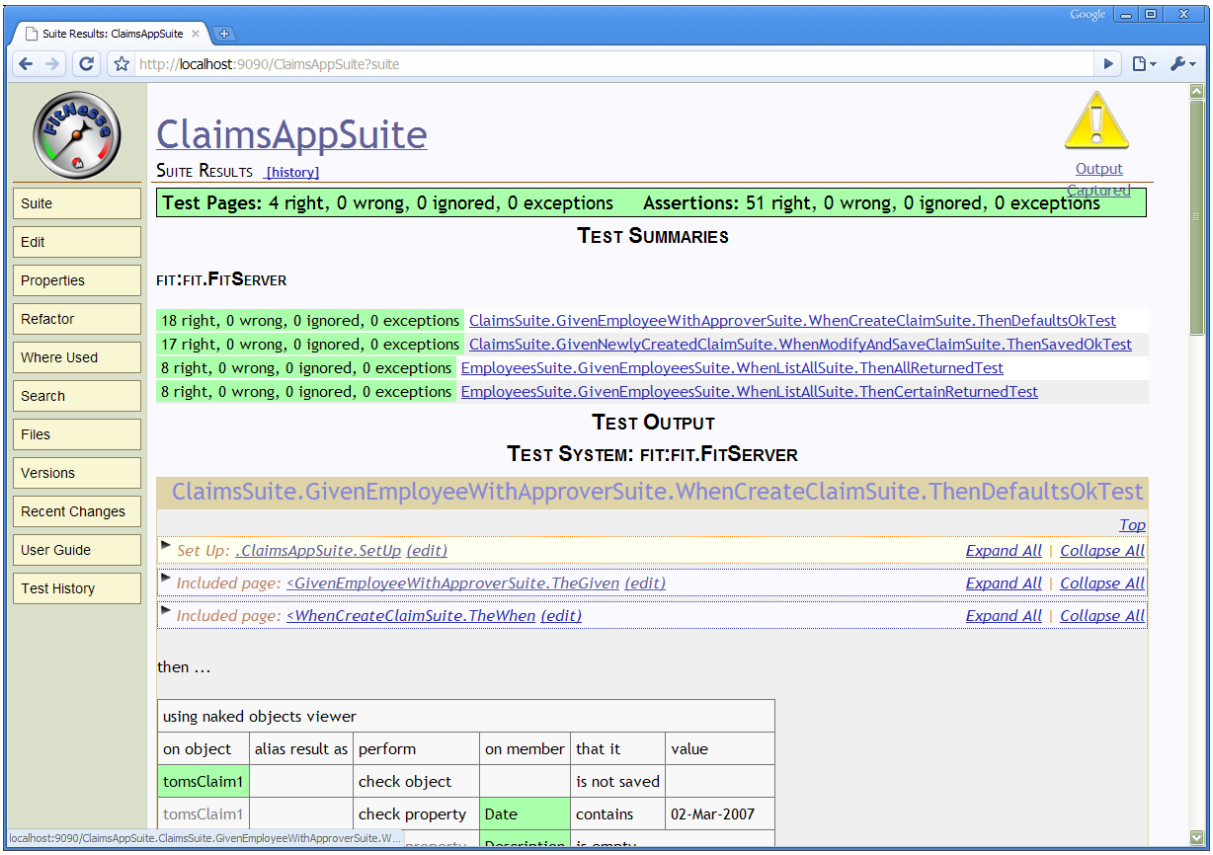
Scenario tests tend to act against a complete system, or sometimes at a subsystem-level. At any rate at a granularity that makes sense to a non-technical businesss person. Compare this to unit testing which exercises the behaviour / method of a single class.

Scenario testing is more usually called "agile acceptance testing", but I find that term rather clumsy. Other names also exist, including behaviour-driven development, and example-driven development. Here we use our own term "scenario testing".

1.2. Introduction to FitNesse

FitNesse is an framework to enable scenario testing. It builds upon the earlier FIT framework, which executes tests expressed in terms of tables. FitNesse sits on top of FIT[1], allowing the tables to be written within a Wiki, and spawning off FIT to run the tests. The results of the tests are shown as annotated tests, as shown below.

[1] In fact, FitNesse as of end 2008 has a replacement test execution architecture, called SLIM.



This makes it easy for non-technical business users to both author new tests, and to view their execution. It also creates an efficient feedback loop; a FitNesse test will "keep on going" even if it hits a failure. Thus the developer can identify several issues and fix them in a single pass.

Another way to think of FitNesse is as a replacement presentation layer, hitting the underlying domain model in the same way that the regular UI would.

If using FitNesse on a "regular" project then the developer writes glue code that in effect take the values out of the wiki page, and use them to interact with the system. They then return a success/failure response which FitNesse then uses to annotate the results page. FitNesse calls this glue code a "fixture". There is some overlap with Naked Objects' own use of that term. However, whereas a Naked Objects fixture is only used to setup the initial state of a test, a FitNesse fixture not only does that but it also is used to execute the test proper.

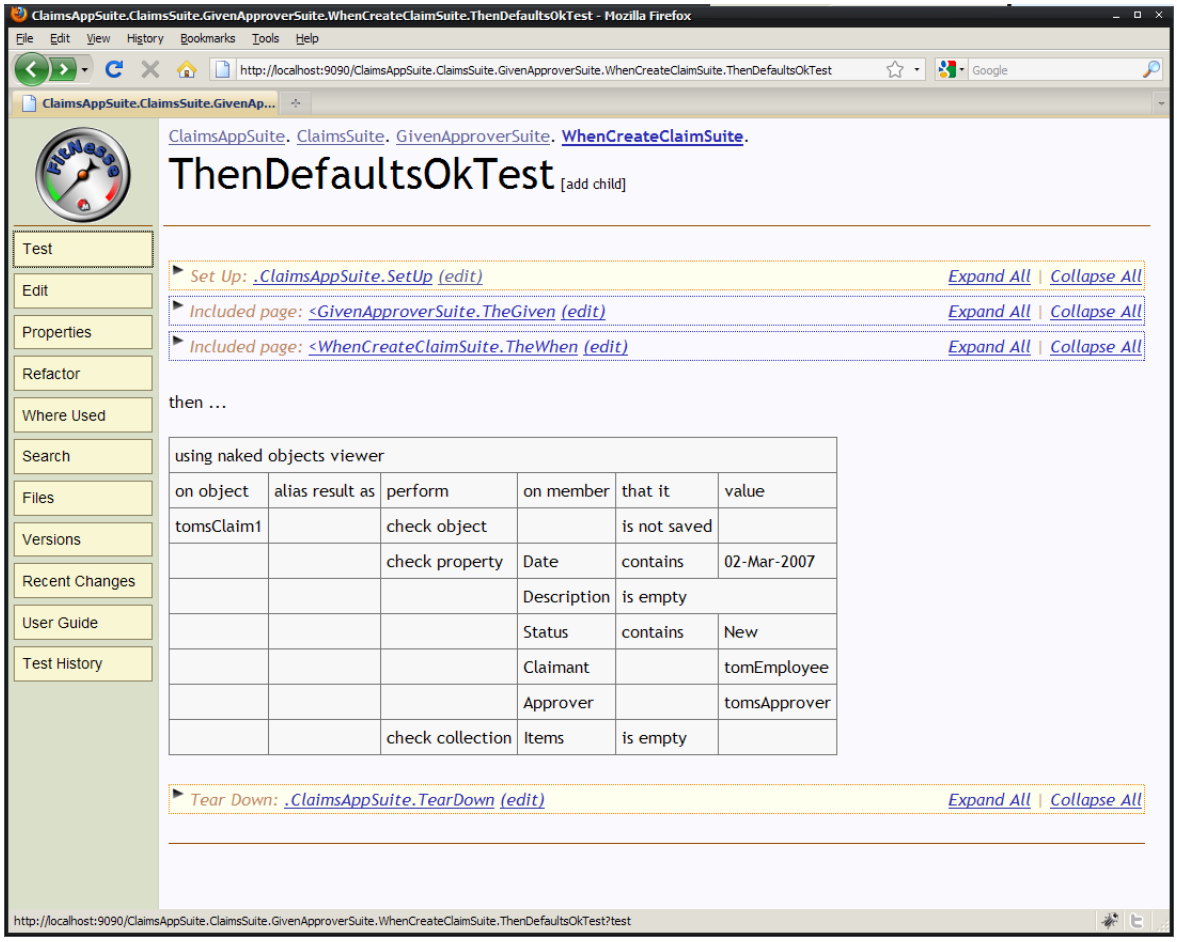
It is relatively straightforward to integrate FitNesse into a continuous integration environment; see Section 3.9, “Set up Continuous Integration” for further details.

1.3. How Tested Objects' Integration Works

Although you could test a Naked Objects application using vanilla FitNesse, this would entail you having to write all the FitNesse fixtures to interact with the domain objects. You would also need to come up with a representation for the tables.

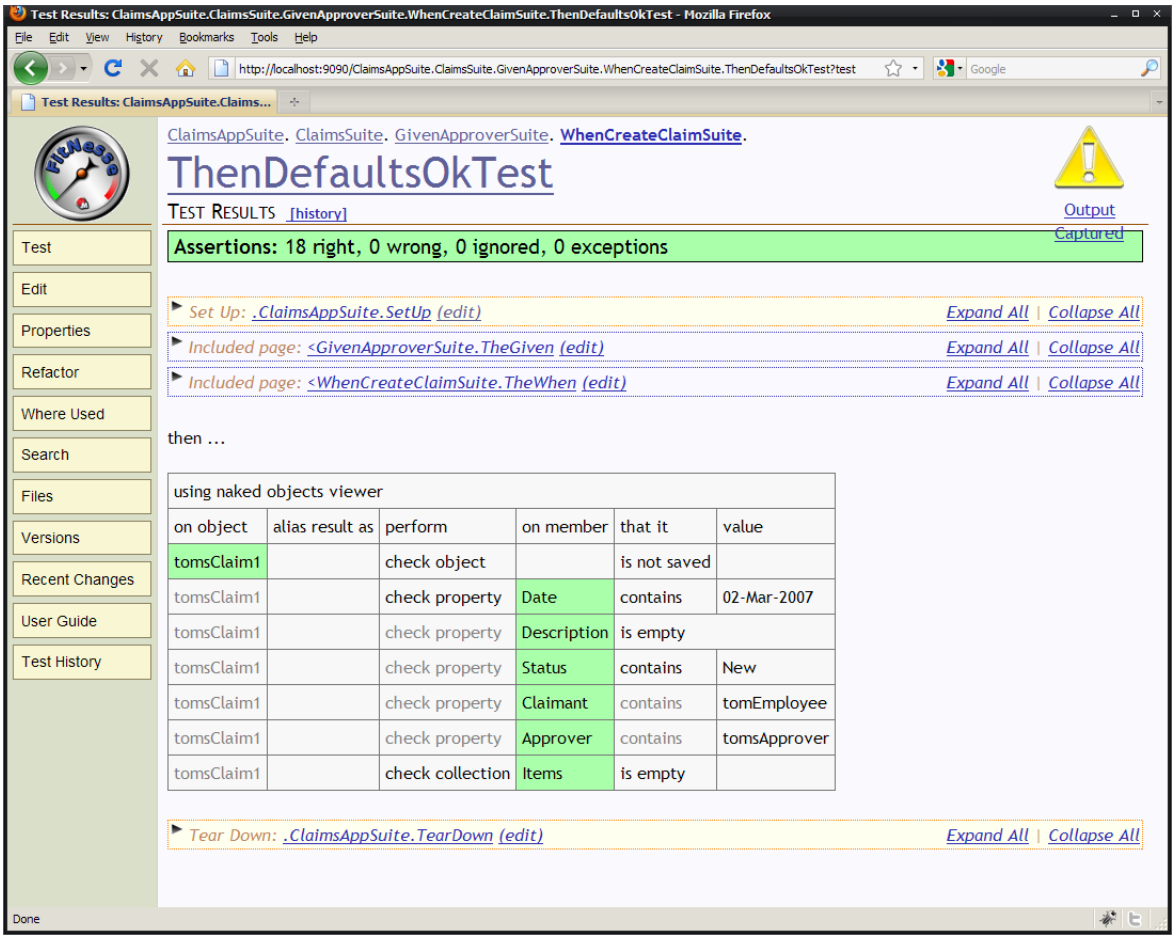
Tested Objects' FitNesse integration is designed to let you use FitNesse to test your Naked Objects application without all this hassle. We can use the same Naked Objects metamodel that is used for the auto-generation of its generic OOUIs to create a set of generic fixtures that interact with the domain object similarly. Using Tested Objects is therefore just a matter of using these fixtures out-of-the-box.

There are many such fixtures, but the one you will use the most often is the UsingNakedObjectsViewer fixture (see the section called “UsingNakedObjectsViewer / UsingNakedObjectsViewerForSetup”). The screenshot below shows it in use for checking assertions at the end of a test (the "Then").



To explain this: the "tomsClaim1" is an alias to an object that has been created or obtained previously. The "perform" column lists the verb to do, in this case all checks. Other things that can be done include invoke actions, set properties and so forth. The "on member" column specifies the property, collection or action of the domain object being interacted with.

The following screenshot shows how the test is annotated after a successful run:



The end-user should be able to follow this test and could, if they want, run through the same set of steps manually themselves. Or, of course, they could just look at the test results.

Chapter 2

Using the FitNesse Archetype

Like Naked Objects, Tested Objects provides a Maven archetype to get you up and running with quickly. Tested Objects' FitNesse archetype runs against any Naked Objects application, and provides a new `xxx-fitnesse` project (where `xxx` is the root artifact Id). This project bundles FitNesse, contains the FitNesse wiki pages that contain the tests (and also a user guide), and allows the FitNesse wiki server to be run.

Out-of-the-box the archetype will generate a set of tests for the example claims application in the `nakedobjects-4.0.x-for-maven.zip` download[1], under `examples/claims`). These are great to try out *Tested Objects* for the first time, and they also provide some ideas for you to structure your own test suite.

The `xxx-fitnesse` project generated by the archetype is intended to be included as a module of the parent pom (`xxx/pom.xml`) --- in Maven parlance, it is a partial archetype. Providing that your run the archetype in the correct directory, Maven will automatically include the newly generated project in the parent project.

2.1. Prerequisites

To use the archetype you'll need to install Maven (and there's a good chance you've done this already if you're using the Naked Objects Maven archetype).

Optionally (and recommended) you can install the m2eclipse plugin for Eclipse IDE (again, there's a good chance you've done this too).

2.2. Running the Archetype

You can either run the archetype from the command line (and then import into Eclipse if using m2eclipse), or run the archetype using m2eclipse straight off.

[1] See http://sourceforge.net/projects/nakedobjects

Running the archetype from the command line

Navigate to the parent directory of the project (that holds the parent pom.xml). The directions here are for the examples/claims project.

TODO: Naked Objects 4.0.0 released examples/claims as org.nakedobjects.distribution:examples-claims. In 4.0.1 (the plan is) for it to be released as org.nakedobjects.examples:claims. In the screenshots that follow I've hacked the 4.0.0 pom to change the groupId and artifactId. However, I've provided the command that kicks the whole thing for both 4.0.0 and 4.0.1

To run the archetype from the command line (against NOF 4.0.0):

```
$ cd $NO_HOME/examples-claims
$ mvn archetype:generate \
  -D archetypeCatalog=http://starobjects.sourceforge.net/m2-repo/snapshot \
  -D archetypeGroupId=org.starobjects.tested.fitnesse \
  -D archetypeArtifactId=archetype \
  -D archetypeVersion=1.0-beta-3-SNAPSHOT \
  -D groupId=org.nakedobjects.distribution \
  -D artifactId=claims-fitnesse \
  -D parentArtifactId=claims \
  -D package=org.nakedobjects.examples.claims.fitnesse \
  -D version=4.0.0
```

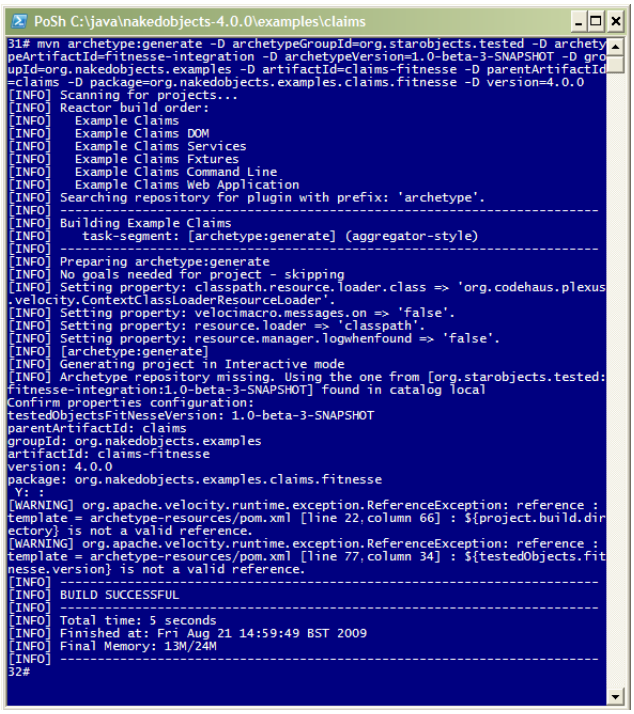
or against NOF 4.0.1:

```
$ cd $NO_HOME/examples-claims
$ mvn archetype:generate \
  -D archetypeCatalog=http://starobjects.sourceforge.net/m2-repo/snapshot \
  -D archetypeGroupId=org.starobjects.tested.fitnesse \
  -D archetypeArtifactId=archetype \
  -D archetypeVersion=1.0-beta-3-SNAPSHOT \
  -D groupId=org.nakedobjects.examples \
  -D artifactId=claims-fitnesse \
  -D parentArtifactId=claims \
  -D package=org.nakedobjects.examples.claims.fitnesse \
  -D version=4.0.1
```

The groupId:parentArtifactId:version tuple should correspond to the parent pom (so, for 4.0.0, that is `org.nakedobjects.distribution:claims:4.0.0`, and for 4.0.1 it is `org.nakedobjects.examples:claims:4.0.1`). Note that the parentArtifactId is not one of the standard Maven properties, instead it is an additional property mandates by the Tested Objects' archetype. (The archetype uses this property to derive the names of some the other projects to include in the dependencies).

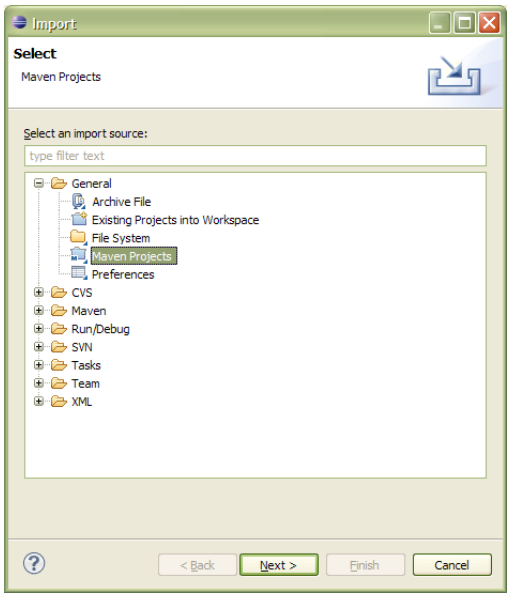
The artifactId then is the artifact of the FitNesse project being generated, so is typically *parentname*-fitnesse (so here: `claims-fitnesse`).

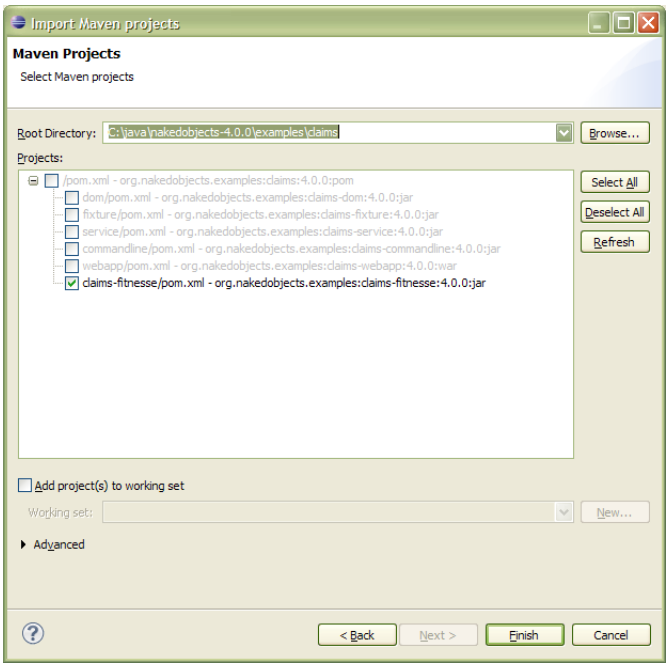
The output is shown below:



This will generate a claims-fitnesse directory, and update the parent pom.xml file.

We can then import into project into Eclipse, using File > Import > Maven





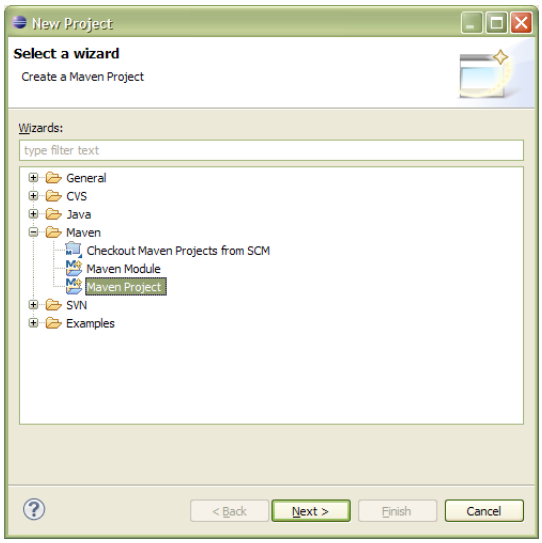
Running the archetype from m2eclipse

As an alternative to running the archetype from the command line, you can instead run the archetype using m2eclipse.

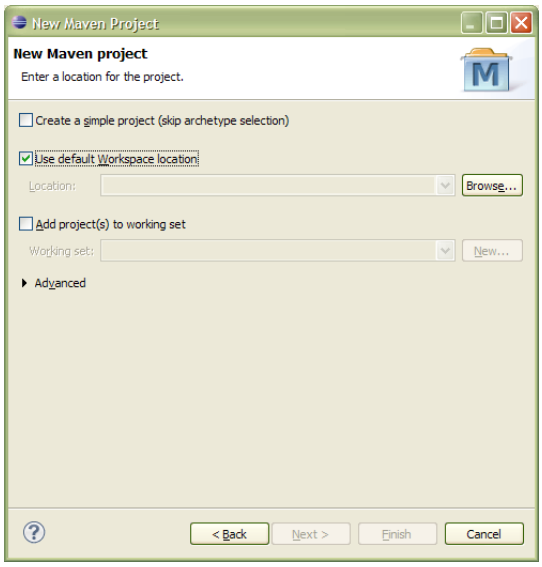
However, m2eclipse does not (seem to) understand partial archetypes, and so the project cannot be generated in a subfolder of the parent project. Instead, I recommend you create the project alongside the parent (eg in `examples/claims-fitnesse`).

TODO: These notes are slightly incomplete; it is also necessary to specify the archetype catalog in the Windows>Preferences. However, m2eclipse 0.9.8 (the latest 'stable' version at the time of writing) fails to pick up snapshot archetypes from remote catalogs. The workaround is either to use the command line once (this will copy the archetype into your local repository) or alternatively achieve the same thing by building the FitNesse code from scratch, as described in the developers' guide.

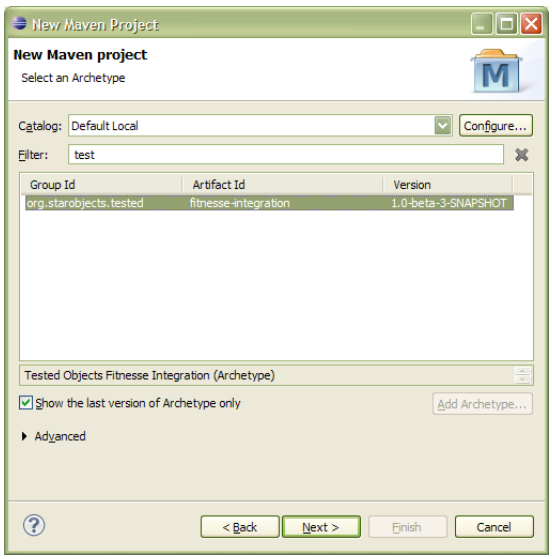
So, use File > New > Project, then Maven > Maven Project to bring up the wizard:



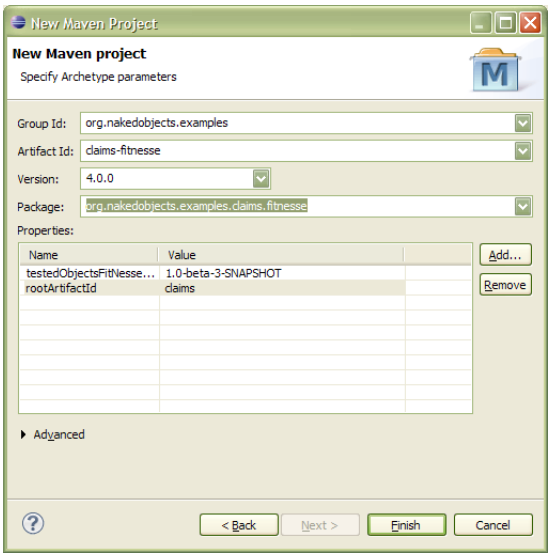
Specify the directory. Note that this cannot be a directory that contains a pom.xml file.



Select the archetype. (Note that the screenshot here shows the archetype in the "Default Local" catalog ... that's because I've built the archetype from source.).



Specify the groupId, artifactId (eg `claims-fitnesse`), version and package. Also specify the parentArtifactId as the parent project's artifactId (eg `claims`):



Hit Finish and you will be in more-or-less the same place as if you had generated the archetype from the command line and then imported. The only thing different is that the new fitnesse project will not be in the parent project's modules.

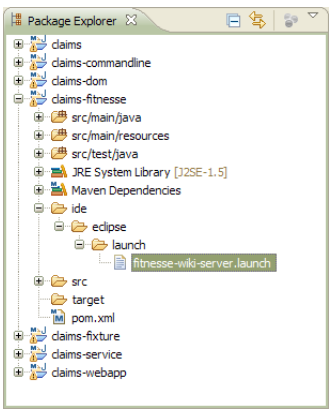
Therefore, navigate to the parent project's `<modules>` region, and add:

```
<modules>
    <module>dom</module>
    <module>fixture</module>
    <module>service</module>
    <module>commandline</module>
    <module>webapp</module>
    <module>../claims-fitnesse</module>
</modules>
```

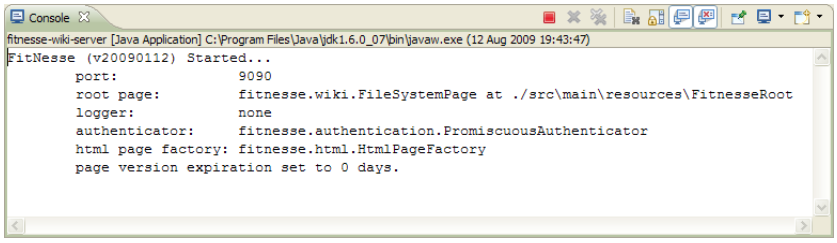
The instructions in the wiki *do* assume that you have done this.

2.3. Starting FitNesse Wiki

Once the fitnesse project has been imported you can start the wiki. Navigate to `ide/eclipse/launch` and there will be a launch configuration:



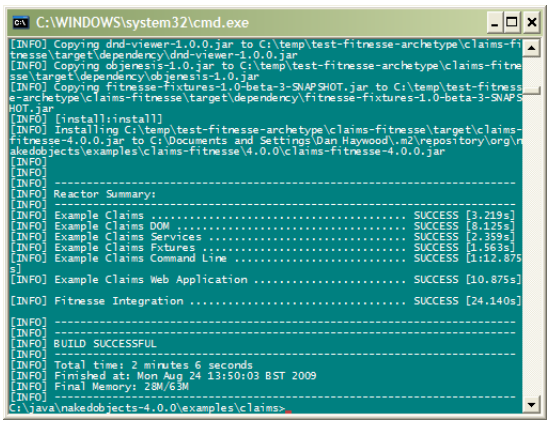
Start by right clicking and selecting run as. The FitNesse wiki should start on port 9090:



2.4. Setting up to Run Tests

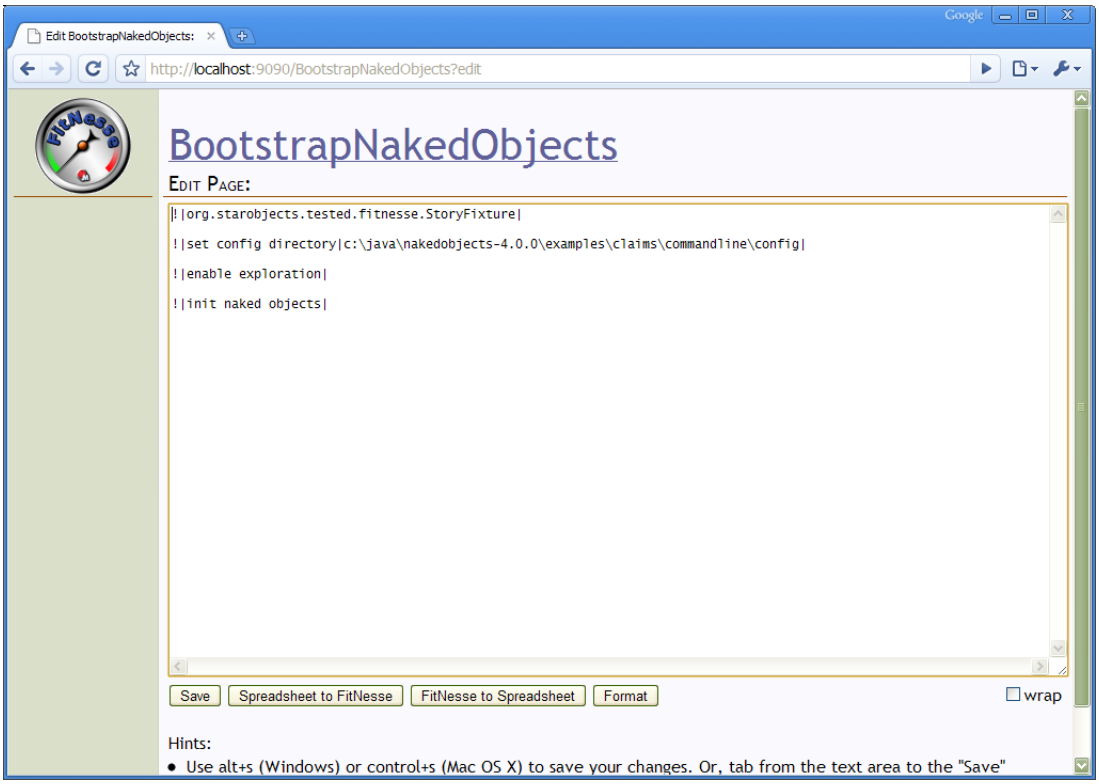
Using your favorite browser, navigate to http://localhost:9090/FrontPage; you should see the introductory page:

Before we can run the generated tests there are a couple of things we need to take care of (both are documented on FrontPage). First, we need to copy the application into a location so that FitNesse can find it. This is done by simply running mvn clean install on the project:



(Note, if for any reason you didn't include the xxx-fitnesse project into the parent project, you'll need to run mvn clean install for the xxx-fitnesse project as well).

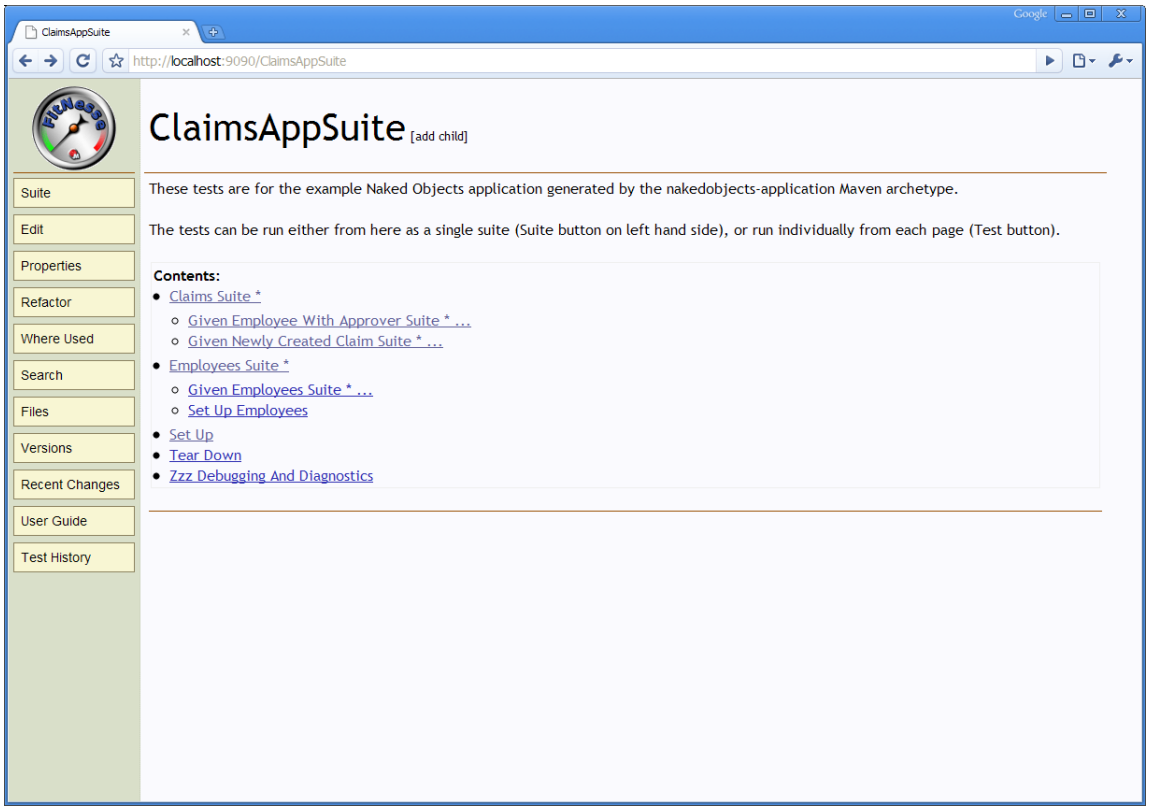
Secondly, the pages generated by the archetype use the `nakedobjects.properties` to bootstrap Naked Objects. By default, this is assumed to be in `../commandline/config/nakedobjects.properties`. If this isn't the case, then edit the BootStrapNakedObjects page as required:



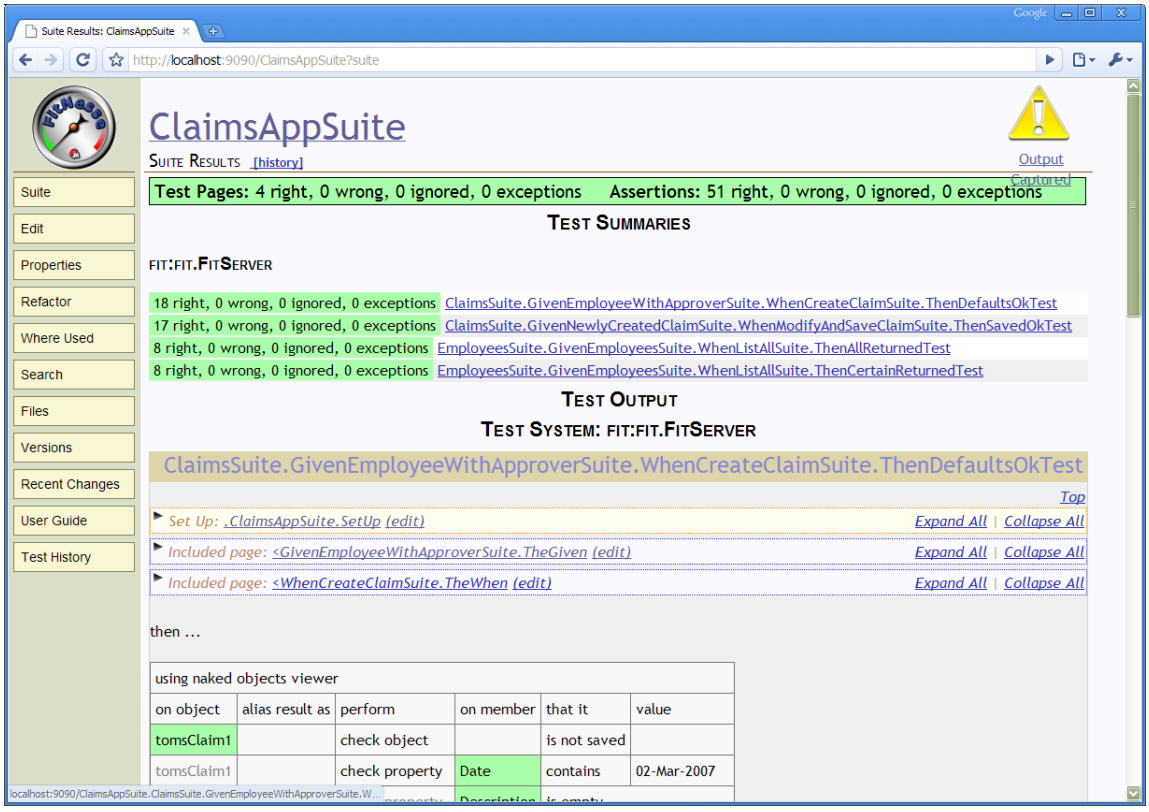
2.5. Running the Tests

If you are trying out the archetype against the examples/claims application then you'll probably want to run the existing tests to check everything works.

The tests are organized into a single application suite, ClaimsAppSuite. This in turn defines two subsuites:



FitNesse will automatically invoke all tests in the hierarchy, therefore just press the Suite button on the left:



If you've run the archetype against your own project though, then you can either delete these ClaimsAppSuites, or leave them around for reference. To ensure that they don't accidentally run, use the Properties button (on the left hand side) and deselect the Suite and Test properties.

Chapter 3

Hints and Tips

This chapter contains a collection of hints, tips and suggestions for writing your own tests.

For further guidance, check out Gojko Adzic's book, Bridging the Communication Gap.

3.1. Separate In-Progress Stories from the Backlog

If you are using an agile methodology then you will be implementing a number of stories per iteration; the remainder will be in a backlog. When you select a story for implementation, create a new page for it in a "CurrentIteration" suite. The objective for the team is therefore to get the entire CurrentIteration suite green.

Other stories that you may have identified but not selected for the iteration can remain in a Backlog suite.

3.2. Use a Story Page to Collect A Set of Scenario Tests

Part of estimating the size of a story includes identifying the acceptance criteria. These can be created as children of the story page as placeholders, so that the story page becomes a suite. The child scenario tests can be fleshed out as required with plain text during the estimation meeting, and with actual FitNesse tests once the iteration starts. The FitNesse !contents instruction will then list all the acceptance criteria for the story.

For the story page itself, the "as a ... I want ... so that... " template is a good way to summarize the intent of the story.

3.3. Organize Completed Stories by Component

Once you have completed an iteration and implements its stories, move those stories out to the relevant component that the story relates to. The scenario tests for stories ultimately *are* the documentation of the

behaviour of the system. A year on you won't remember (and won't care) which iteration you implemented a story, you'll be searching for it by the component whose behaviour you want to understand.

3.4. Structure your test using Given/When/Then

A standard template for organizing structuring tests is given/when/then[1]:

- given ... the system is in this particular state
- when ... this interesting thing happens
- then ... these are the consequences

This structure is readily understood by non-technical business users, and helps them (and the team) focus on the point of the test.

In terms of mechanics, one approach is to put the "given" into the setup page for a test, with the "when" and the "then" in separate pages. Alternatively, as the archetype does (see Chapter 2, *Using the FitNesse Archetype*), you could separate out the "given", the "when" and the "then" into a hierarchy of pages.

- Separating out the "given" from the rest of the test makes it easy to include that given in other tests (discussed further in Section 3.6, “Factor out common "Given"s”), and it also allows us to run up the viewer to inspect the setup (see Section 3.5, “Using the RunViewer fixture”).
- Separating out the "then" from the "when" makes it easy to identify the individual post conditions. A new requirement might mean only the addition of a new post condition. A downside is that the "given" and "when" will be run for each post-condition, leading to longer test turn-around times.

3.5. Using the RunViewer fixture

One reason that the archetype (Chapter 2, *Using the FitNesse Archetype*) separates out the "given", "when" and "then" is so that the "given" - which is often the hardest part to get setup - can be verified independently from the rest of the test.

To do this, we can use the RunViewer fixture (see the section called “RunViewer”). This will run up the drag-n-drop viewer at the specified point in the test; a visual equivalent of `System.out.println()`, really. We can therefore take the Given page and add a RunViewer fixture at the end.

Note that to do this you must temporarily mark the Given page as a test page.

3.6. Factor out common "Given"s

Just like code, tests need to be actively managed, because if the tests become hard to maintain, they'll end up being deleted. In fact, we probably should take even more care with the tests than the code if they represent the primary documentation of the behaviour of the system.

In terms of size, the "given" is far larger than either the "when" or the "then", and therefore this is the area where tests can quickly become unmaintainable. So instead, factor out your givens into separate pages,

[1] As first described, I believe, by Dan North in a blog post, Introducing BDD.

and then use FitNesse's !include directive to assemble the pages you need (as done by the archetype, Chapter 2, *Using the FitNesse Archetype*).

The names of these pages should also follow a declarative style, see Section 3.7, “Use a Declarative Style for Page Names”.

3.7. Use a Declarative Style for Page Names

When factoring out "given"s (see Section 3.6, “Factor out common "Given"s”), or indeed when writing the "when"s and the "then"s, use a declarative style for the pages. The page should describe what it does, not how it does it.

For example, a good page would be "SetUpCountries". It's clear that this will set up all `Country` reference data classes. This could be included into a "SetUpReferenceData" page. For transaction data, we could have a page "JoeBloggsCustomer"; another one again could be "JoeBloggsFiveOrders".

3.8. Run against a real database

The Tested Objects' integration exercises the Naked Objects domain model as configured in nakedobjects.properties (as per the SetUpConfigDirectory fixture, ???). Out-of-the-box, of course, Naked Objects uses an in-memory object store, and so there are no issues running one scenario test against another.

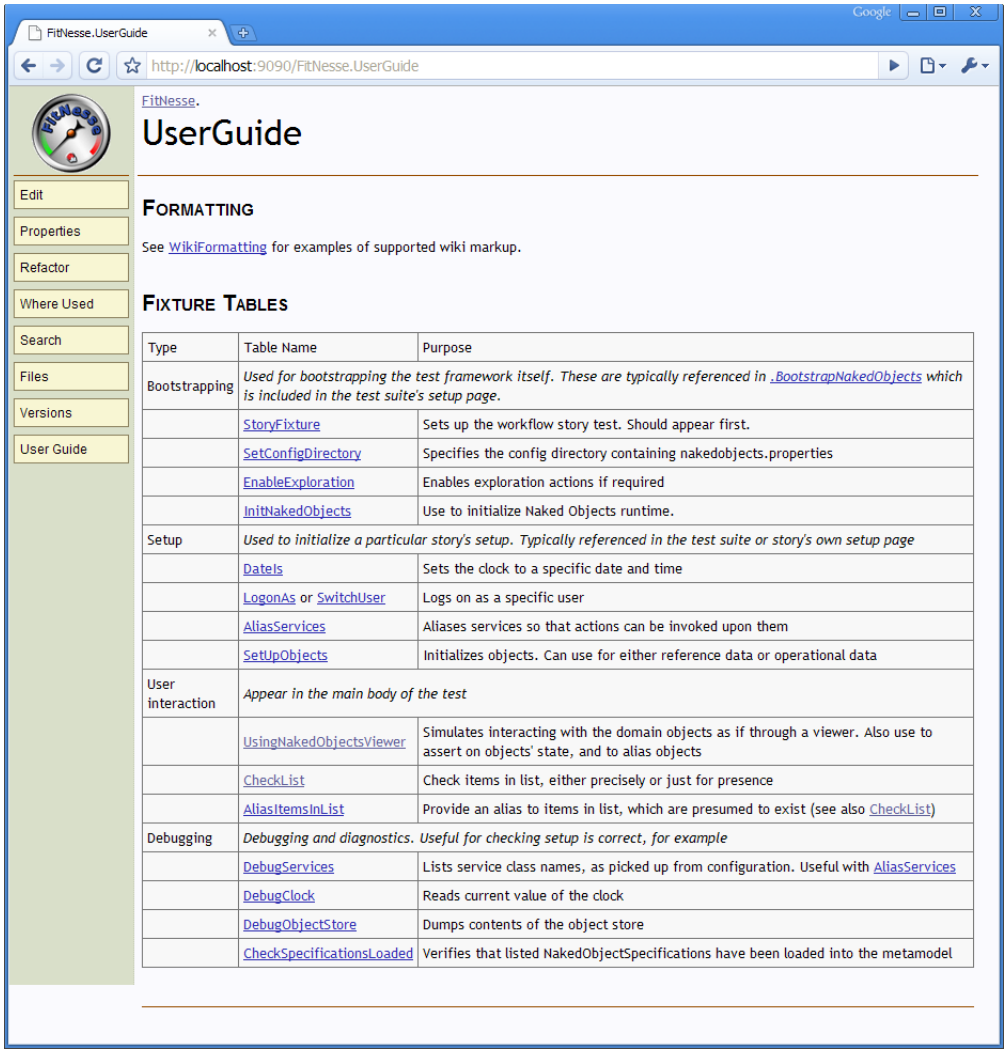
You can if you want though configure Naked Objects to go against a real database, for example by using the Hibernate-based object store provided by the JPA Objects sister project. In this case you will need to ensure that you reset the database at the end; Tested Objects doesn't provide any fixtures to help you though, so you are on your own here. An alternative might be to use JPA Objects against HSQLDB configured to for in-memory use. This would let you verify your database mappings, but without no need to tear anything down.

3.9. Set up Continuous Integration

Since Tested Objects is a Maven application, it is easy enough to configure it to run under a CI server, such as Hudson. If you google around you should be able find a way to publish the FitNesse test results through Hudson.

Appendix A. Reference (Online User Guide)

For convenience when writing tests the FitNesse wiki pages created by the Maven archetype (see Chapter 2, *Using the FitNesse Archetype*) also include an online user guide:



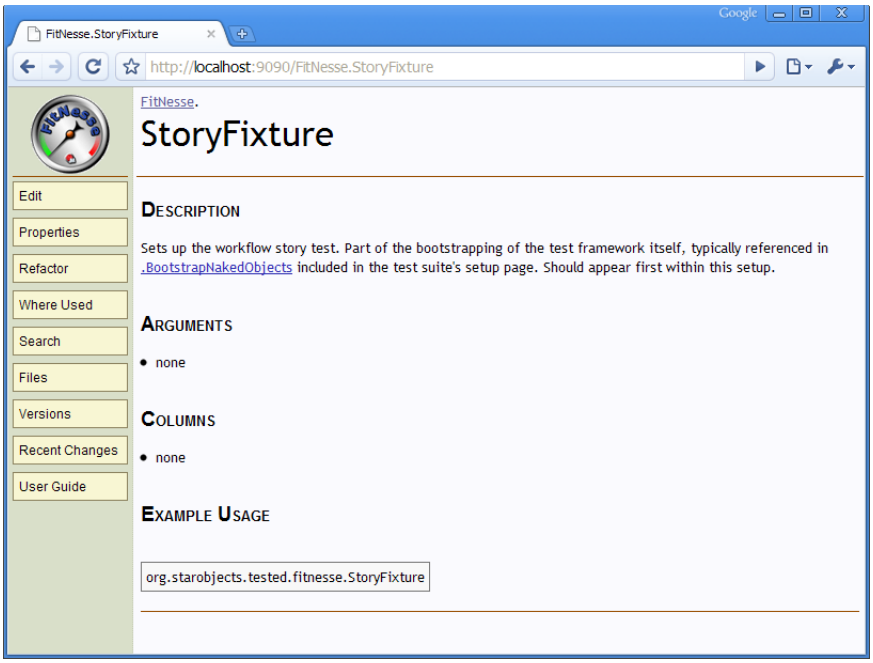
Rather than repeat the text here, this reference guide just consists of screenshotsof the various pages.

A.1. Bootstrapping

The bootstrapping fixtures are used to bootstrap the test framework itself. These are typically referenced in a "BootstrapNakedObjects" page, included in the test's setup page. One option is to use the FitNesse SetUp page.

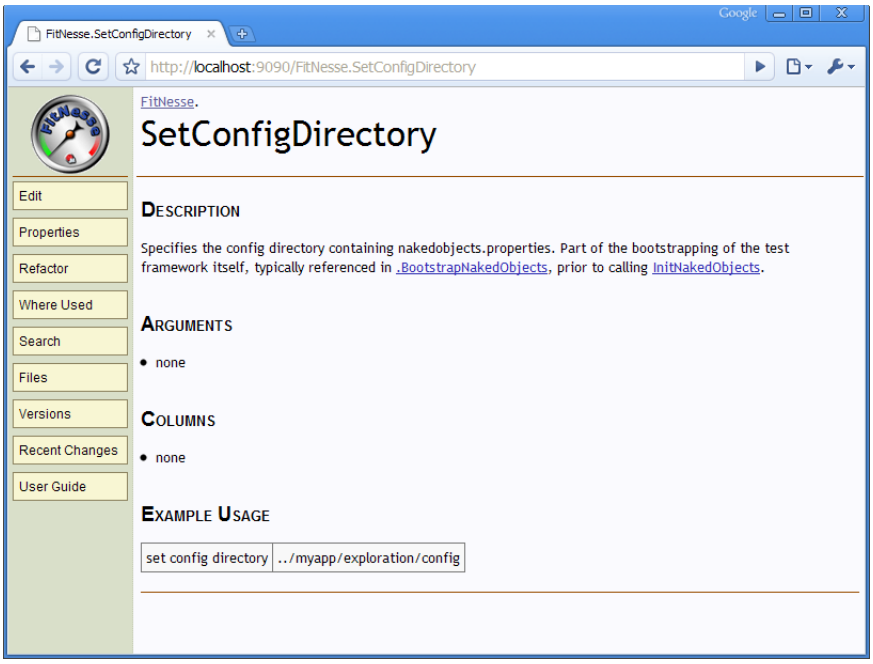
StoryFixture

Sets up the workflow story test. Boilerplate, should always be the first FitNesse fixture included in a page.



SetConfigDirectory

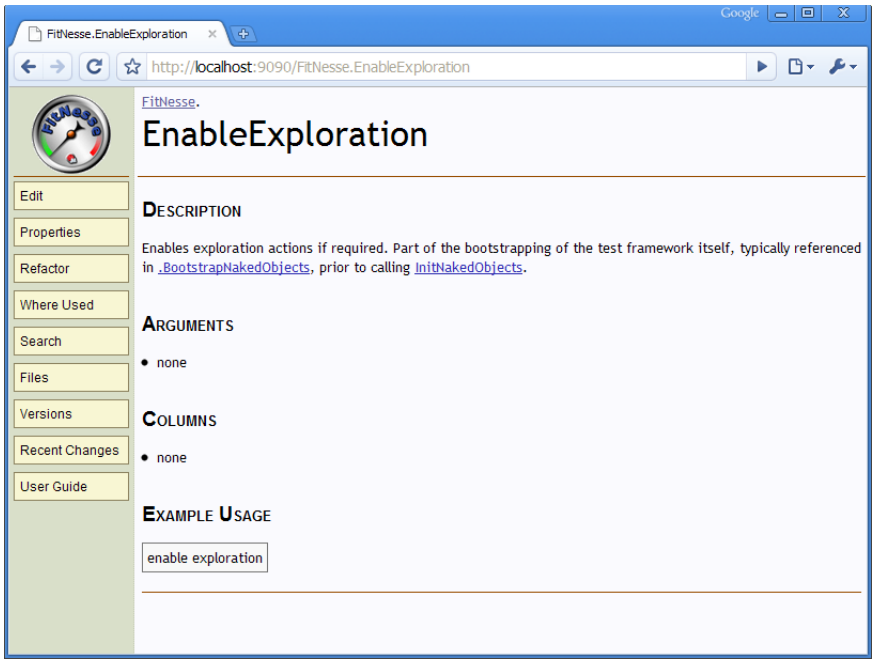
Specifies the config directory containing `nakedobjects.properties`. Called after StoryFixture (see the section called “StoryFixture”), and before InitNakedObjects (see the section called “InitNakedObjects”). The DebugServices fixture (the section called “DebugServices”) can be used to debug the set of services specified (after Naked Objects has been initialized).



EnableExploration

Enables exploration actions if required. Should be called before InitNakedObjects (see the section called “InitNakedObjects”).

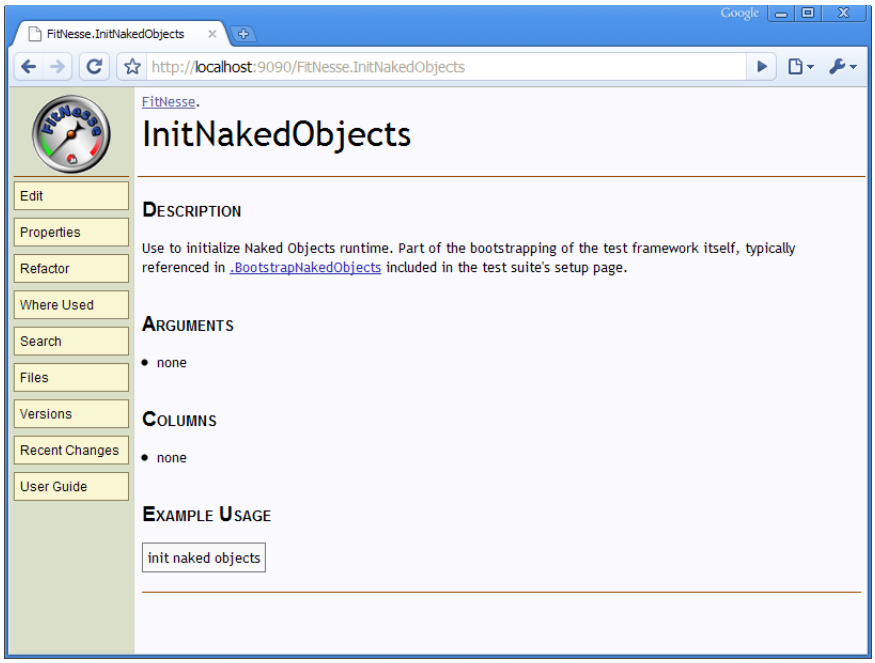
Note that when using the DnD or HTML viewers, exploration mode means that there is no need to logon. For FitNesse tests though you should specify who to login as, see the section called “LogonAs”.



InitNakedObjects

Initializes the Naked Objects runtime using the services specified through the UseConfigDirectory fixture (see the section called “SetConfigDirectory”).

The CheckSpecificationsLoaded fixture (the section called “CheckSpecificationsLoaded”) can be used to check which classes have been located from the services as a result of initialization.

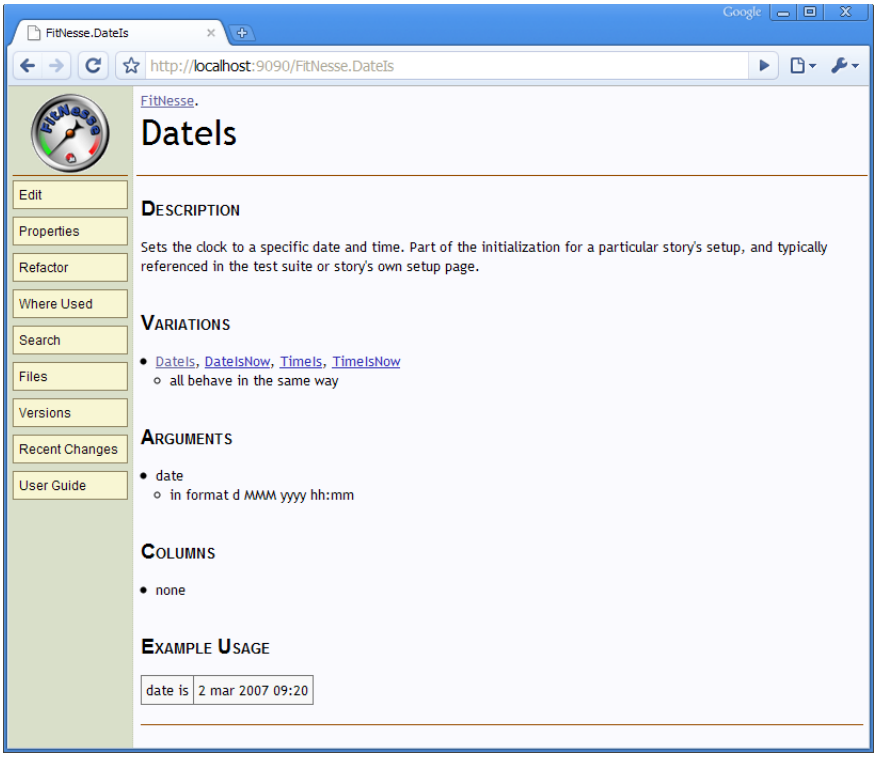


A.2. SetUp

The setup fixtures are used to specify the running application for a particular story's setup. Specifically, this means setting up the services that define the application, the effective date and the effective user. It also allows the setup of arbitrary objects (typically reference/static data objects; for transactional objects see the section called “UsingNakedObjectsViewer / UsingNakedObjectsViewerForSetup”).

DateIs

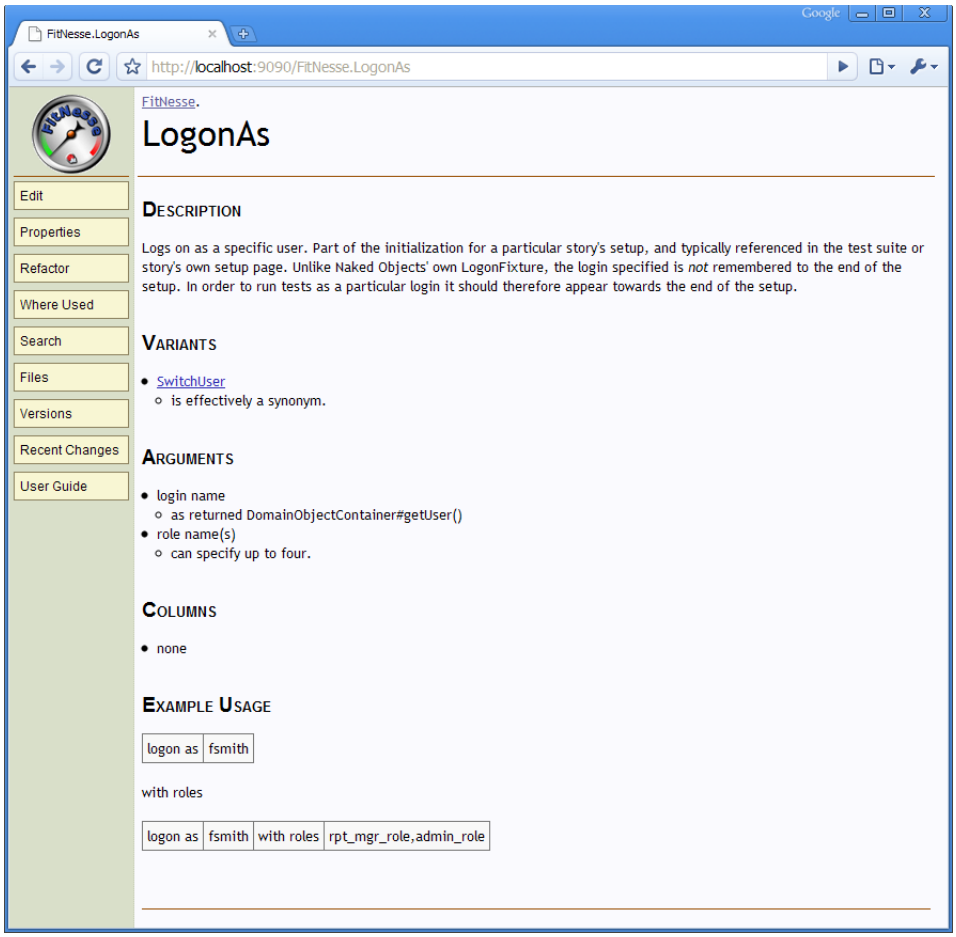
Sets the clock to a specific date and time. This installs the `FixtureClock` as the implementation of the `Clock` singleton (in the applib). If this fixture is not called, then the default system clock is used, which gets the time from the host computer. The DebugClock fixture (the section called “DebugClock”) can be used to verify the clock state.



LogonAs

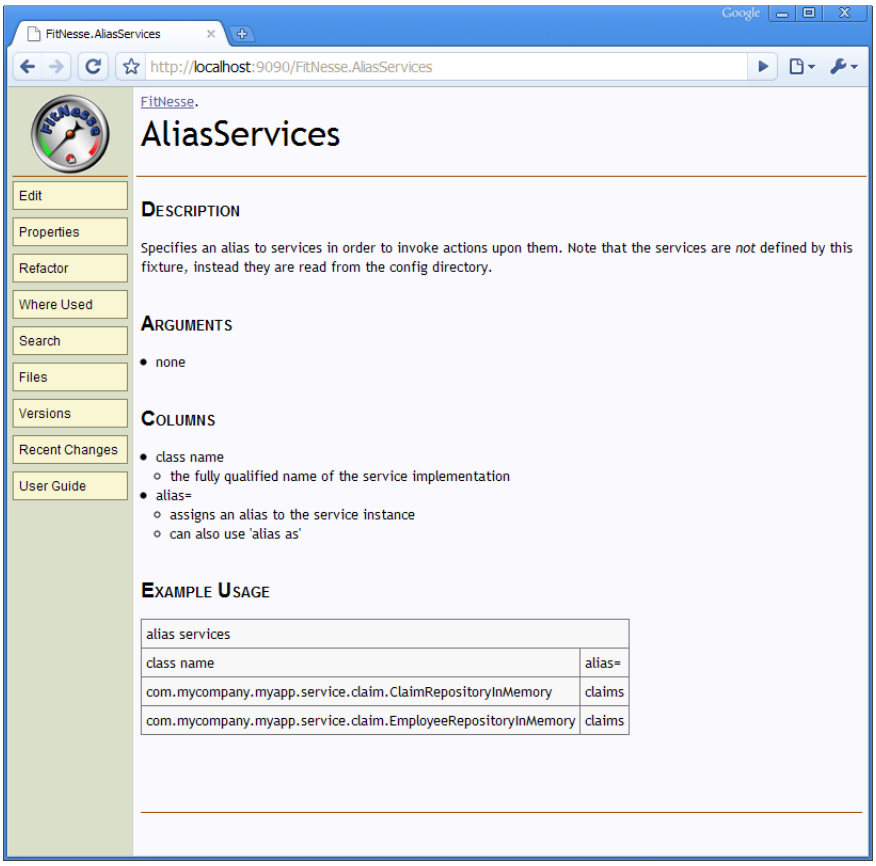
Logs on as a specific user.

Unlike Naked Objects' own `LogonFixture`, the login specified is not remembered to the end of the setup. In order to run tests as a particular login it should therefore appear towards the end of the setup.



AliasServices

Specifies an alias to services in order to invoke actions upon them. Note that the services are *not* defined by this fixture; for that see SetConfigDirectory fixture, section the section called “SetConfigDirectory”. See also DebugServices (the section called “DebugServices”) to verify the services that have been identified.



SetUpObjects

Initializes objects. Typically used for immutable reference/standing data objects). Can also be to setup used for transaction/operational data objects (though UsingNakedObjectsViewerForSetup, the section called “UsingNakedObjectsViewer / UsingNakedObjectsViewerForSetup”, is preferable). The DebugObjectStore fixture (the section called “DebugObjectStore”) can be used to check the state of objects created.

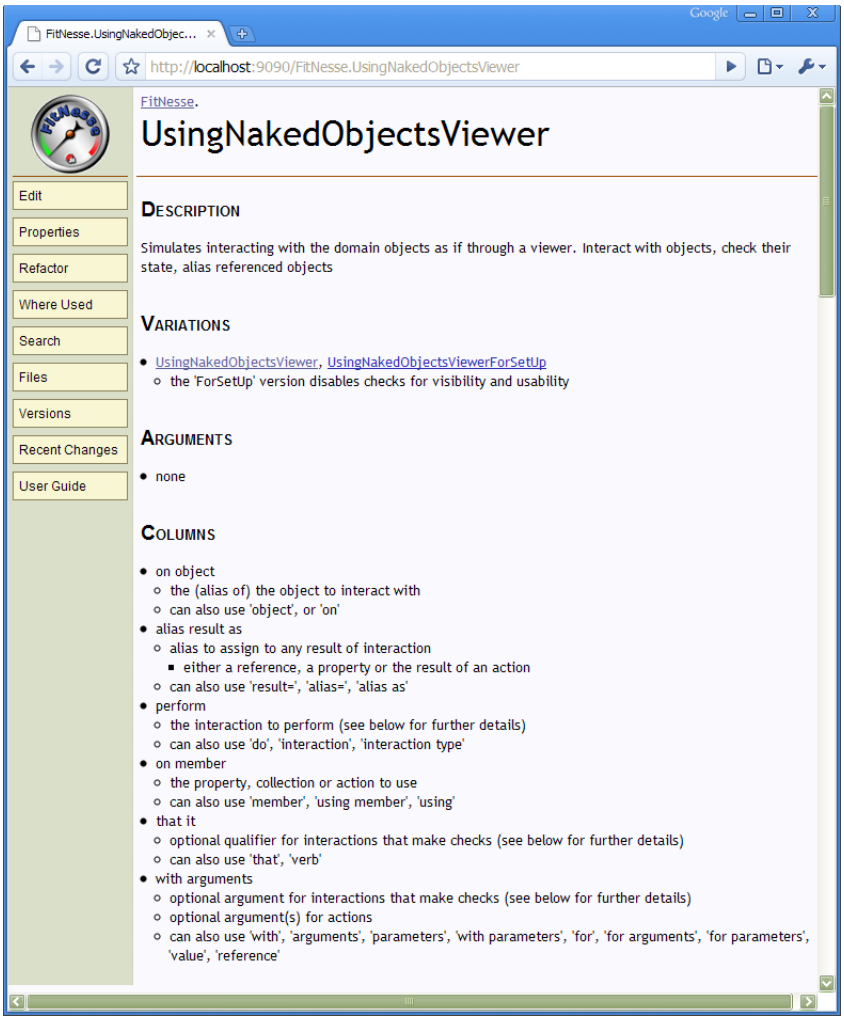
A.3. User Interaction

User interaction fixtures appear in the main body of the test, for the "given" (to setup the rest of the state of the system, typically transactional objects), for the "when" (the interaction being tested) or the "then" (assertions on the state after the interaction being tested).

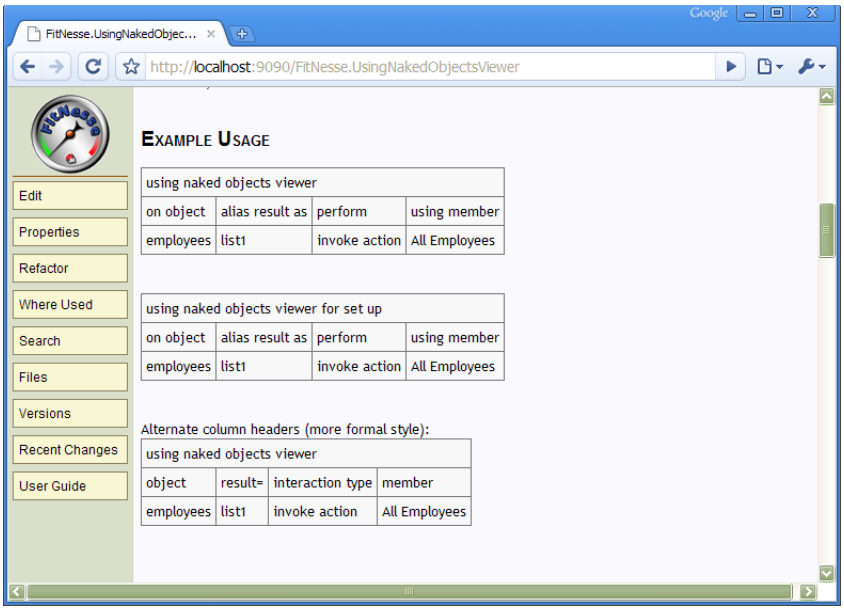
UsingNakedObjectsViewer / UsingNakedObjectsViewerForSetup

Simulates interacting with domain objects as if through a viewer. Interact with objects, check their state, alias referenced or returned objects.

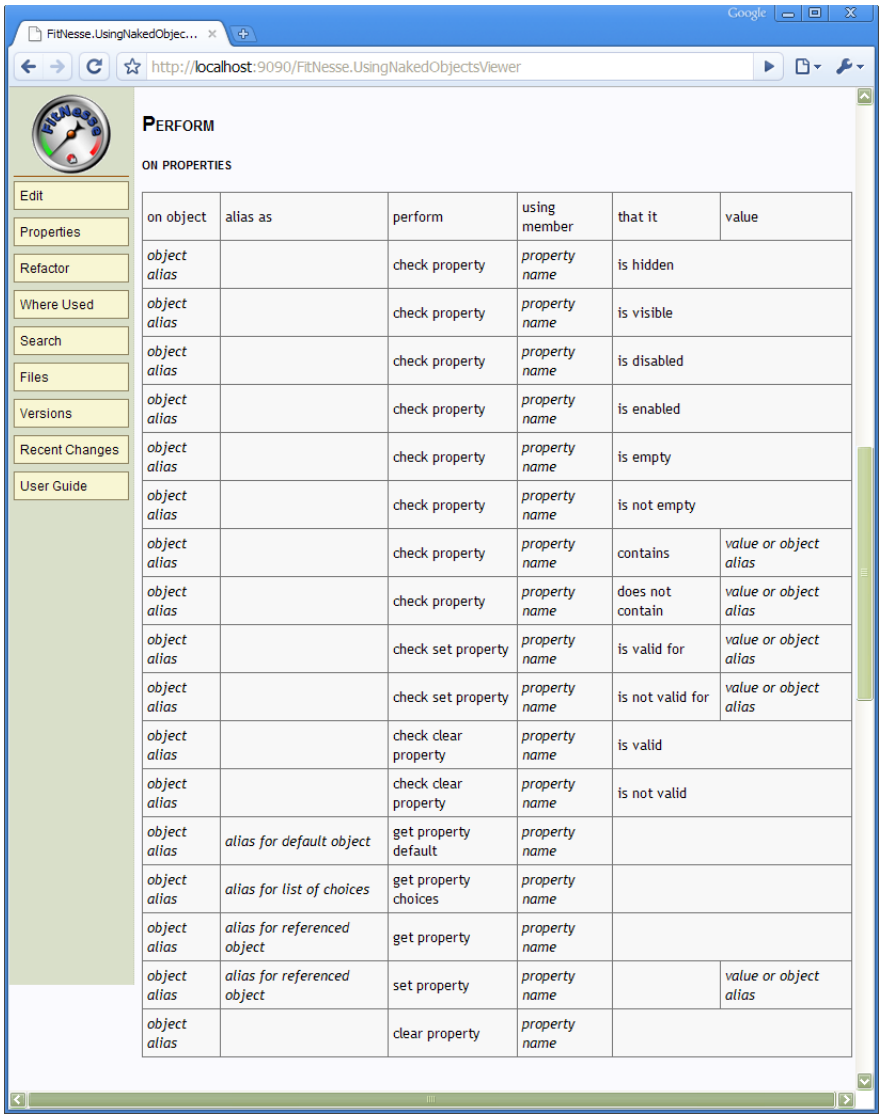
The "ForSetup" version disables checks for visibility and usability, making it easier to reuse functionality for setting up objects prior to a test scenario (the "given"). The DebugObjectStore fixture (the section called “DebugObjectStore”) can be used to check the state of objects created.



Example usage:



On properties:

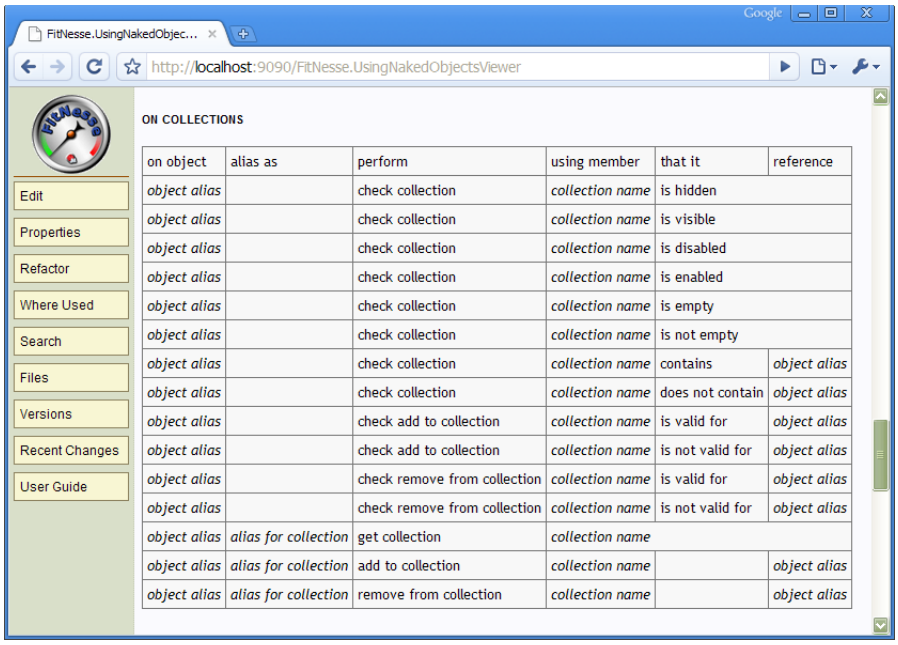


PERFORM

ON PROPERTIES

on object	alias as	perform	using member	that it	value
object alias		check property	*property name*	is hidden	
object alias		check property	*property name*	is visible	
object alias		check property	*property name*	is disabled	
object alias		check property	*property name*	is enabled	
object alias		check property	*property name*	is empty	
object alias		check property	*property name*	is not empty	
object alias		check property	*property name*	contains	*value or object alias*
object alias		check property	*property name*	does not contain	*value or object alias*
object alias		check set property	*property name*	is valid for	*value or object alias*
object alias		check set property	*property name*	is not valid for	*value or object alias*
object alias		check clear property	*property name*	is valid	
object alias		check clear property	*property name*	is not valid	
object alias	*alias for default object*	get property default	*property name*		
object alias	*alias for list of choices*	get property choices	*property name*		
object alias	*alias for referenced object*	get property	*property name*		
object alias	*alias for referenced object*	set property	*property name*		*value or object alias*
object alias		clear property	*property name*		

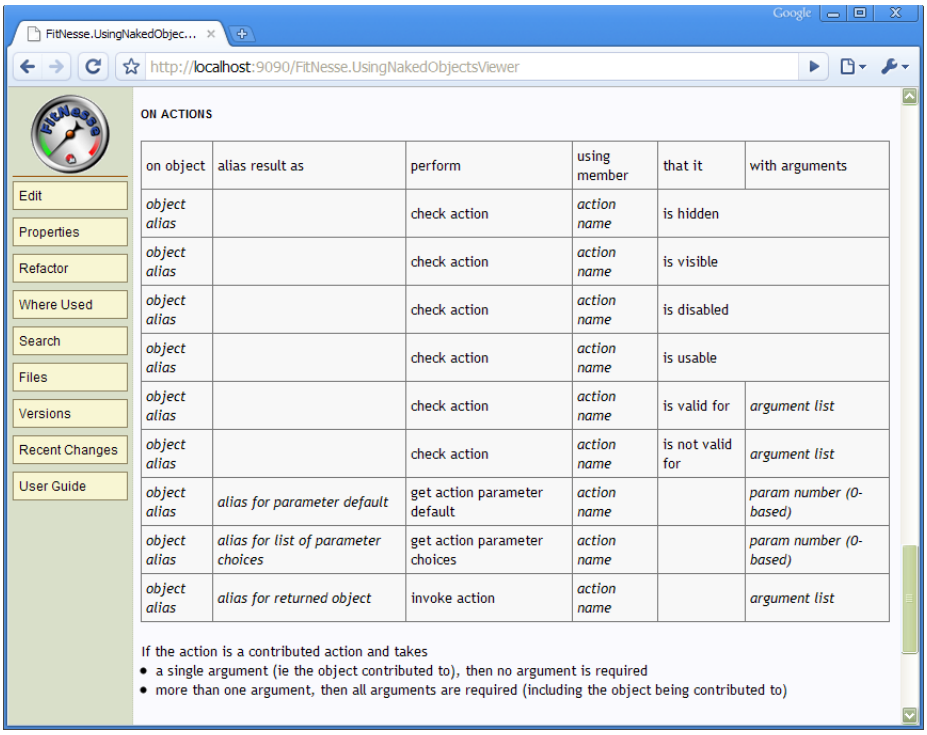
On collections:



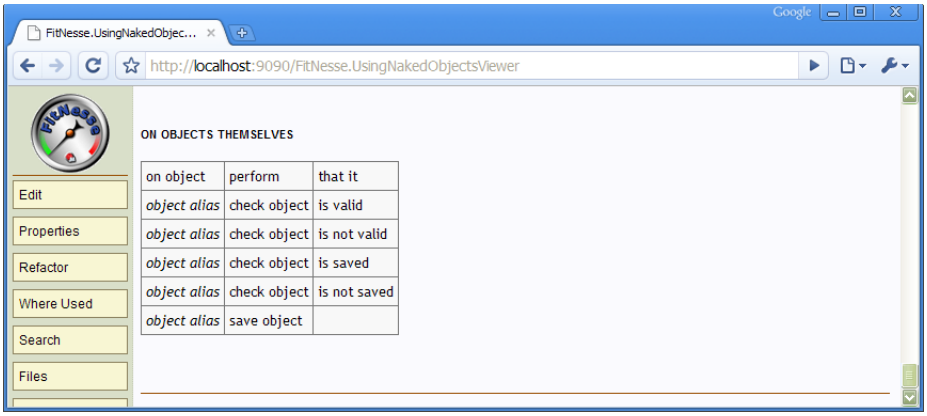
ON COLLECTIONS

on object	alias as	perform	using member	that it	reference
object alias		check collection	*collection name*	is hidden	
object alias		check collection	*collection name*	is visible	
object alias		check collection	*collection name*	is disabled	
object alias		check collection	*collection name*	is enabled	
object alias		check collection	*collection name*	is empty	
object alias		check collection	*collection name*	is not empty	
object alias		check collection	*collection name*	contains	*object alias*
object alias		check collection	*collection name*	does not contain	*object alias*
object alias		check add to collection	*collection name*	is valid for	*object alias*
object alias		check add to collection	*collection name*	is not valid for	*object alias*
object alias		check remove from collection	*collection name*	is valid for	*object alias*
object alias		check remove from collection	*collection name*	is not valid for	*object alias*
object alias	*alias for collection*	get collection	*collection name*		
object alias	*alias for collection*	add to collection	*collection name*		*object alias*
object alias	*alias for collection*	remove from collection	*collection name*		*object alias*

On actions:



On objects:

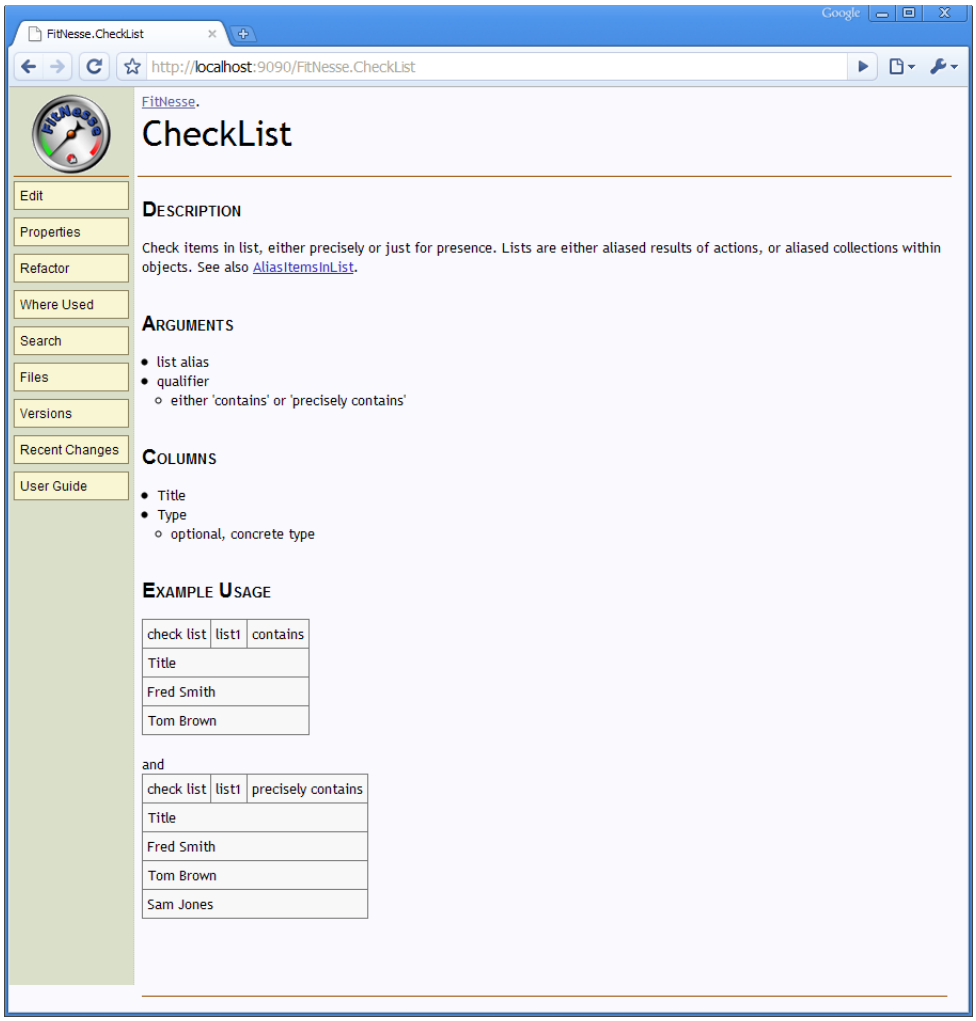


CheckList

Check items in list, either precisely or just for presence, using their title. Lists are either aliased results of actions, or aliased collections within objects.

Typically used in the "Then", though can be helpful as a way of confirming/documenting a "Given".

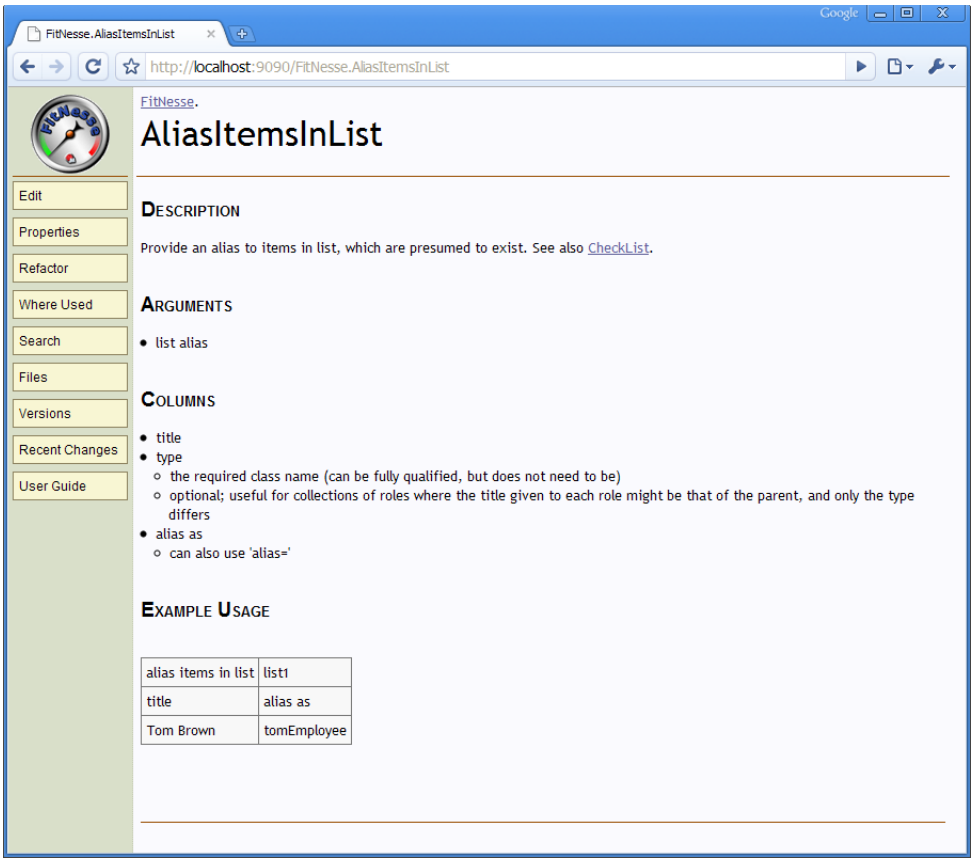
See also AliasItemsInList (the section called “AliasItemsInList”), which also performs an implicit check (will fail if the objects are not in the list) and aliases them for further use.



AliasItemsInList

Allows an alias to be associated with items in a list. The list items are located by their title, and are presumed to exist. This fixture can therefore also be used as a way of checking for presence of items in a list (similar to CheckList, the section called “CheckList”).

Typically used both in the "Given" (to simplify writing the rest of a test).

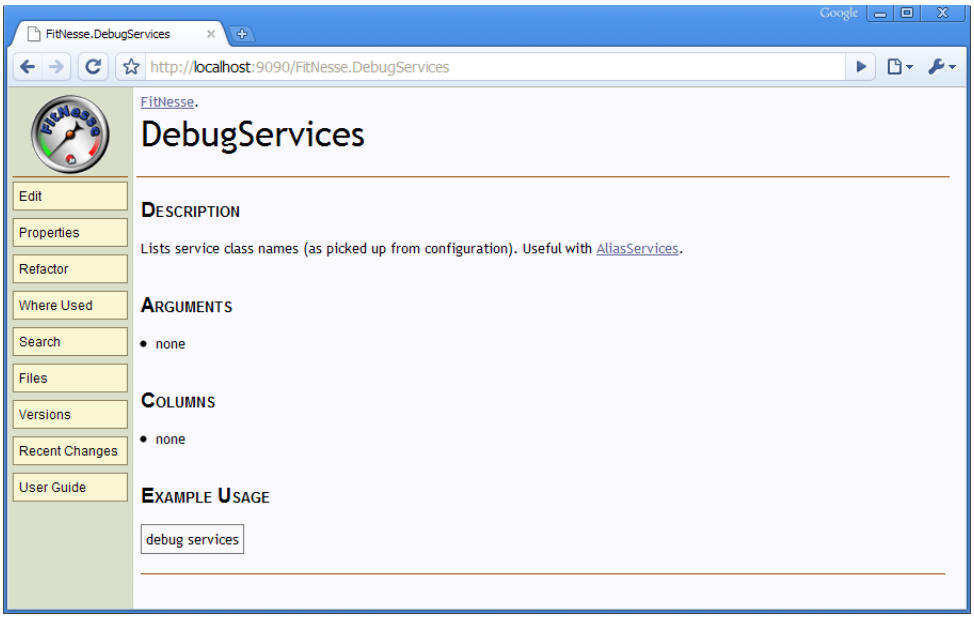


A.4. Debugging

Debugging and diagnostics. Useful for checking setup, for example.

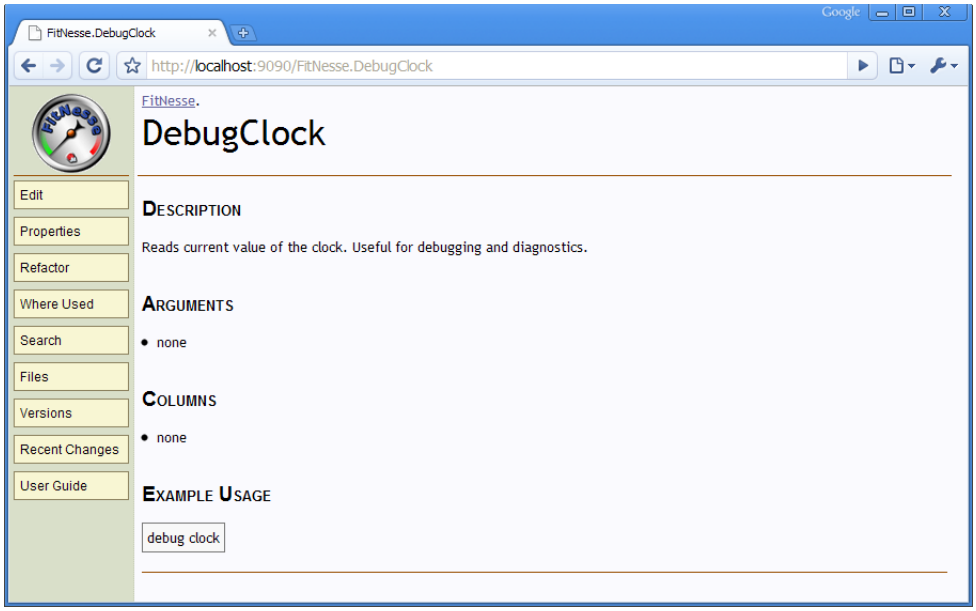
DebugServices

Lists service class names, as picked up from configuration. Useful with AliasServices (see the section called “AliasServices”).



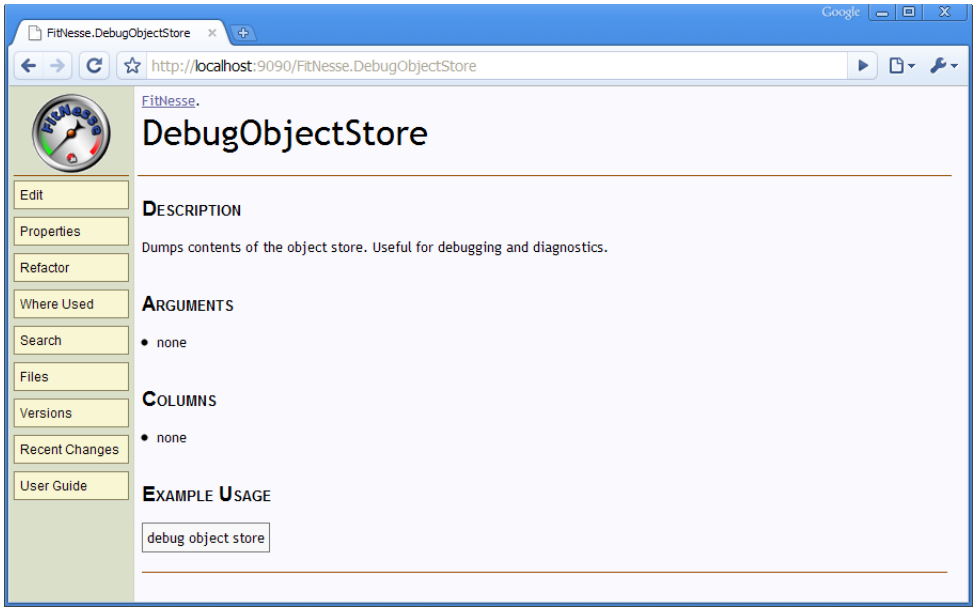
DebugClock

Reads the current value of the clock. Useful for debugging and diagnostics.



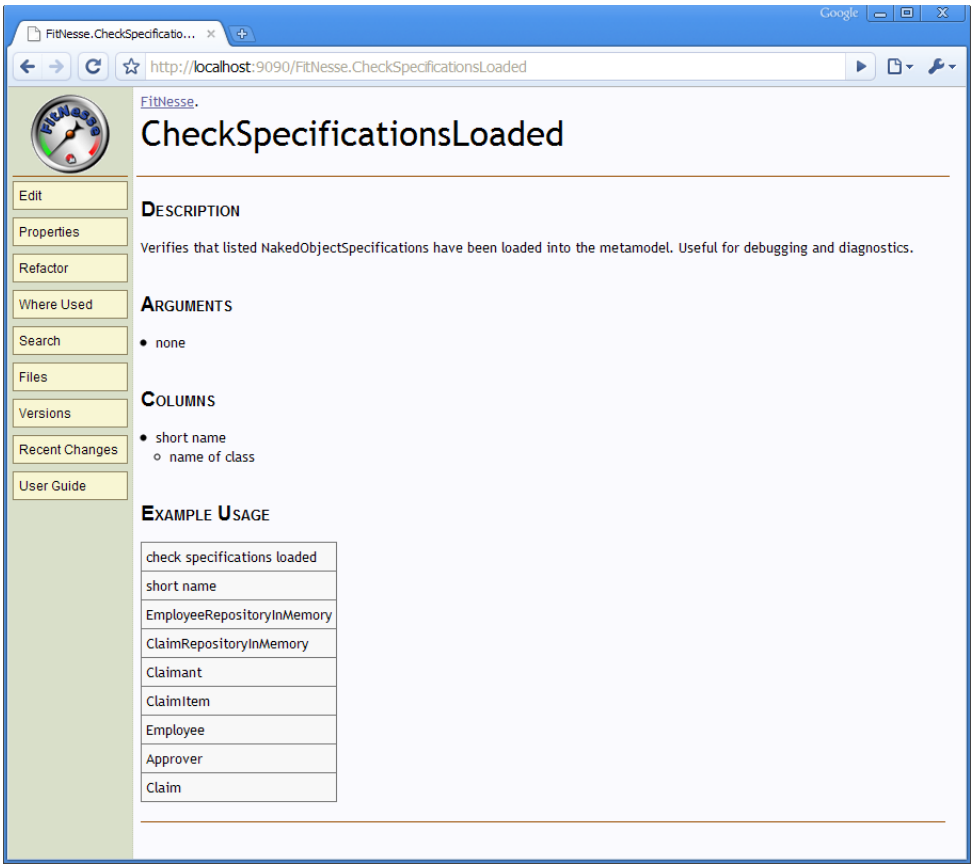
DebugObjectStore

Dumps the contents of the object store. Useful for debugging setup (through SetupObjects, the section called “SetUpObjects”, and UsingNakedObjectsViewerForSetup, the section called “UsingNakedObjectsViewer / UsingNakedObjectsViewerForSetup”).



CheckSpecificationsLoaded

Verifies that listed NakedObjectSpecifications have been loaded into the metamodel.



RunViewer

Runs up the DnD viewer with the current state of the objects. This is a great way to inspect the state of the system, for example if a test is failing and you can't see why.

Of course, other alternatives are writing unit tests and debugging with break points. But failing scenario tests usually arise because of an integration issue between two different bits of the system. Being able to have a "poke around" can be invaluable.

A.5. Tear Down

The opposite of setting up...

ShutDownNakedObjects

This fixture shuts down the Naked Objects runtime, releasing memory and so on. A good place to put this is in the test's TearDown page (which could be inherited).