Groovy Objects Users Guide

Groovy Domain Object Support for Naked Objects 4.0.x

Version 0.1-SNAPSHOT

Preface ..... **v**
1. Introduction **1**
2. Configuring your (Maven) Project **3**
2.1. Structure of a Naked Objects Application 3
2.2. Updating the Main Project 4
2.3. Updating the DOM Project 5
2.4. Updating the Fixture and Service Projects 8
2.5. Updating the Commandline and/or Webapp Project 8
3. Writing Domains Object in Groovy **9**
3.1. (Optional) Superclass 10
3.2. Properties and Collections 10
3.3. Title & Icon 11
3.4. Creating and Persisting Objects 11
3.5. Callbacks 11
3.6. Annotations 12
3.7. Supporting Methods 13
4. Writing Fixtures in Groovy **15**

Preface

Groovy Objects is a sister project to Naked Objects, allowing domain objects to be written in the Groovy dynamic language as well as in Java.

Groovy 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

This chapter explains what Groovy Objects is for, and a little about how it works.

The Naked Objects framework enables design-driven applications to be rapidly developed and optionally deployed, automatically providing a runtime-generated OOUI for the domain objects. *Naked Objects* is written in Java, and normally the domain objects that make up the application are also written in Java. These objects are basically pojos; *Naked Objects* provides a number of annotations and defines a number of coding conventions so that business rules and constraints can be picked up by the framework, and to expose behaviour in the UI over-and-above simple CRUD operations.

Naked Objects performs its magic by building a metamodel of the underlying domain objects, and uses this to build the OOUI. The process is very similar to the way that ORMs such as Hibernate work, but rather than reflecting the domain objects into the persistence layer, it reflects them into the presentation layer.

Groovy (as I'm sure you know) is an alternative language for writing code to run on the JVM. It offers a number of dynamic language features, as well reduced syntax clutter (eg for properties) along with programming constructs such as closures.

What *Groovy Objects* provides is the ability to write domain objects in Groovy and then run on top within *Naked Objects*. Because Groovy source files are ultimately compiled down into Java bytecode, *Naked Objects* is able (with a little bit of tweaking) to build up its metamodel and run as normal. What *Groovy Objects* does is perform the tweaking in how the metamodel is built up.

This user guide explains how to configure your project to develop using Groovy, and provides some guidance on how to follow the *Naked Objects* coding conventions while programming in Groovy. We generally recommend you develop your domain applications using Apache Maven, and *Groovy Objects* itself is packaged as a Maven module. The details provided focus solely on how to update a Maven-based project; we also explain how to configure your application within an IDE.

Chapter 2

Configuring your (Maven) Project

Groovy Objects is provided as a Maven module. In this chapter we explain the configurations steps necessary to update your Maven-based domain application, and how to configure a common IDE so you can program in Groovy with Naked Objects.

2.1. Structure of a Naked Objects Application

The typical structure for a *Naked Objects* Maven-based application (and the one you'll end up with if you use *Naked Objects*' Maven application archetype) is:

- `app`

  Main (parent) module, whose `pom.xml` references the submodules

- `app/dom`

  Domain object model, plus interfaces for services, repositories and factories

- `app/service`

  Implementation of services, repositories and factories

- `app/fixture`

  Fixtures, used to seed (in-memory) object store when running in exploration/prototype mode

- `app/commandline`

  Bootstrap for running from the command line (typically, the DnD viewer or HTML viewer)

- `app/webapp`

  Packaging and running as a web application

The application is normally run either from the *commandline* project, using the `org.nakedobjects.runtime.NakedObjects` main class (where the viewer to use is specified using

the `--viewer` flag), or from the *webapp* project (picking up `web.xml` for bootstrapping as a web application),

2.2. Updating the Main Project

There is no Groovy code in the `main` project, but what we do here is to define the version, and where necessary configuration, of dependencies used by the application's submodules. We have classpath dependencies and build dependencies, so both are defined.

There are three things to specify:

- which version of the Groovy runtime to use
- which version of GMaven to use. GMaven is the Groovy maven plugin that we use to compile Groovy, hosted at codehaus.org. By default GMaven specifies a version of the Groovy runtime to compile against, but this can be overridden
- which version of *Groovy Objects* to use. Groovy Objects is not itself written in Groovy, but it does have a dependency on Groovy runtime. We want this to be in sync.

To start with, we specify the versions of these different components, by adding the following to the `pom.xml`:

```
<properties>
    <groovy.version>1.7.2</groovy.version>
    <gmaven.version>1.2</gmaven.version>
    <gmaven.runtime>1.7</gmaven.runtime>
    <groovyobjects.version>0.1-SNAPSHOT</groovyobjects.version>
</properties>
```

The `gmaven.runtime` must be compatible with the `groovy.version`. Typically, if the `groovy.version` is `x.y.z`, then the `gmaven.runtime` will be simply `x.y`. The documentation on GMaven providers for further details; running `mvn groovy:providers` is a good place to start.

> Note: *Groovy Objects* also has a dependency on the Groovy runtime, albeit in a very minor way. This dependency is marked optional in order to allow your application to specify its own Groovy runtime version which if needed can be different from that needed by *Groovy Objects*.

Next, we define how we to compile Groovy code, using the GMaven plugin. We specify the plugin, the version, and its configuration; this goes in `<build>/<pluginManagement>`:

```
<build>
    <pluginManagement>
        <plugins>
            ...

            <plugin>
                <groupId>org.codehaus.gmaven</groupId>
                <artifactId>gmaven-plugin</artifactId>
                <version>${gmaven.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
```

```
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <providerSelection>${gmaven.runtime}</providerSelection>
            </configuration>
        </plugin>

    </plugins>
  </pluginManagement>
</build>
```

Finally, we define the dependencies to *Groovy Objects* and to Groovy. This goes in `<dependencyManagement>/<dependencies>`:

```
<dependencyManagement>
    <dependencies>
        ...

        <!-- Groovy Objects -->
        <dependency>
            <groupId>org.starobjects.groovy</groupId>
            <artifactId>gapplib</artifactId>
            <version>${groovyobjects.version}</version>
        </dependency>

        <dependency>
            <groupId>org.starobjects.groovy</groupId>
            <artifactId>gmetamodel</artifactId>
            <version>${groovyobjects.version}</version>
        </dependency>

        <!-- Groovy -->
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-all</artifactId>
            <version>${groovy.version}</version>
        </dependency>

    </dependencies>
</dependencyManagement>
```

2.3. Updating the DOM Project

The *dom* project is the place where the bulk of your domain objects live. Since these are now written in Groovy, we need to ensure that they are compiled.

Source folders

First off, we must separate our Groovy code from any Java code. To do this, create the following folders:

- `src/main/groovy`
- `src/test/groovy`

It's important to create both of these folders (even if you aren't planning on writing any unit tests ;-) ... the IDE integration that we describe below insists upon it.

Updating the Maven POMs

First, we update the POM our domain objects are compiled using the Groovy compiler. Add the following into `<build>`:

```
<build>
    <plugins>

        <plugin>
            <groupId>org.codehaus.gmaven</groupId>
            <artifactId>gmaven-plugin</artifactId>
        </plugin>

    </plugins>
</build>
```

Next, we add dependencies both to *Groovy Objects*' own applib and to Groovy itself:

```
<dependencies>
    ...

    <dependency>
        <groupId>org.starobjects.groovy</groupId>
        <artifactId>gapplib</artifactId>
    </dependency>

    <dependency>
        <groupId>org.codehaus.groovy</groupId>
        <artifactId>groovy-all</artifactId>
    </dependency>

</dependencies>
```

The *Groovy Objects* applib brings in a transitive dependency to the *Naked Objects* applib along one or two helper classes (of which more in Chapter 4, *Writing Fixtures in Groovy*). In the future the applib may be expanded to include other helper classes or new annotations provided by *Groovy Objects* itself.

Finally, a small workaround. *Naked Objects* picks up icons for domain classes from the classpath, with the Java compiler doing the job of copying these icons from `src/main/resources` into target. The Groovy compiler does not seem to do this. We therefore ensure that the Java compiler runs by adding a small dummy Java class. It could go anywhere, but the `images` package is probably the best location:

```
package images;

/**
 * workaround to force Java compiler to kick in and copy images over to target classpath
 */
class Dummy {}
```

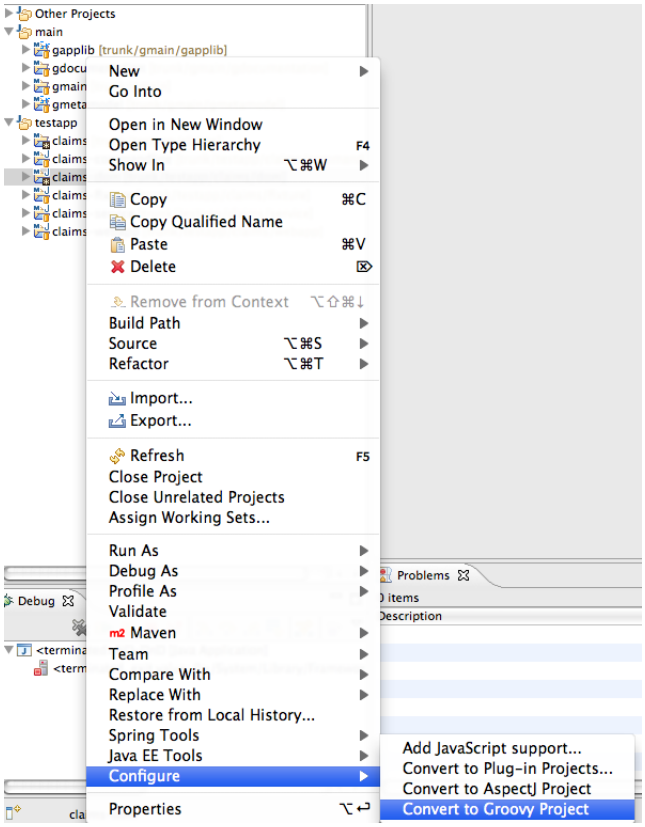
At this point you should be able to build your project from the Maven command line. Let's see how to add in IDE support.

Configuring Eclipse IDE

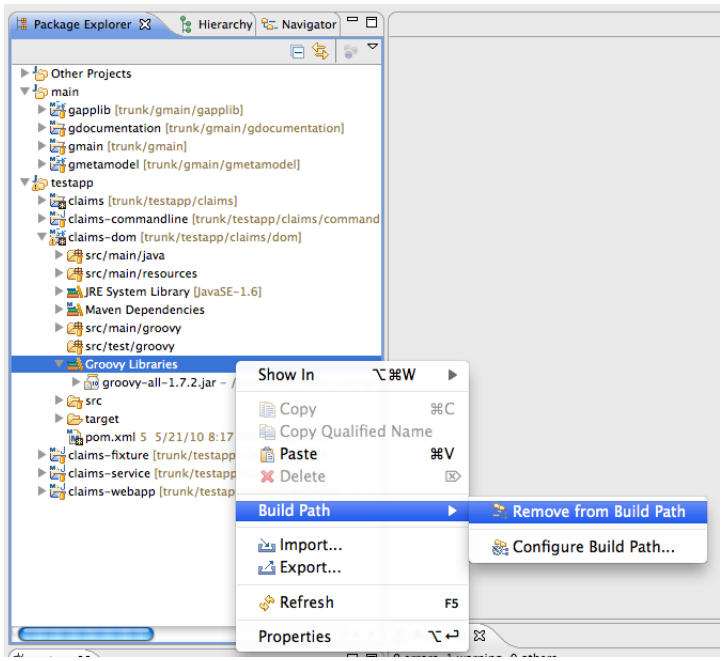
If you are using Eclipse IDE, then you can add in Groovy support using the GroovyEclipse plugin.

To start off with, install the plugin into your IDE using the standard Eclipse update mechanisms. There's a detailed walkthrough on the Groovy wiki if you need it.

Next, enable the Groovy nature in your *dom* project:



Doing this brings in Groovy editor and compiler support (technically: the Groovy nature is added to the project). But it also adds in a reference to the Groovy runtime to our classpath, which we don't need because we already have courtesy of Maven. Therefore, remove the *Groovy Libraries* classpath library:



And that should do the trick.

Other IDEs

If you use another IDE, please let us know what the steps are to configure it, and we'll update this documentation.

2.4. Updating the Fixture and Service Projects

As well as using Groovy for the *dom* project, you may well want to use it for the *fixture* project and the *service* implementations project. In which case, just follow the same steps as the described for the *dom* project.

2.5. Updating the Commandline and/or Webapp Project

Finally, we need to update the *commandline* and/or *webapp* project (depending on how you intend to bootstrap your application). First, we add a dependency to *Groovy Objects*' `gmetamodel` module, in `<dependencies>`:

```
<dependencies>
    ...

    <dependency>
        <groupId>org.starobjects.groovy</groupId>
        <artifactId>gmetamodel</artifactId>
    </dependency>

    ...
</dependencies>
```

Then, update the `nakedobjects.properties` config file as follows:

```
nakedobjects.reflector.facets.include=org.starobjects.groovy.gmetamodel.RemoveGroovyMethodsFacetFactory
```

This little piece of magic alters the way that *Naked Objects* builds up its metamodel of the domain objects. Specifically what it does is to filter out the various methods that the Groovy compiler adds behind the scenes.

In most circumstances that should be enough. If you have very deep hierarchies of domain classes, then you may also need to add in:

```
starobjects.groovy.depth=NNN
```

where NNN should be the depth of the class hierarchy. The default is 5, so in many cases there won't be any need to add this key. (The reason this is needed is that the Groovy compiler seems to generate a set of methods for each level of the class hierarchy).

And you should be good to go.

Chapter 3

Writing Domains Object in Groovy

Naked Objects uses convention over configuration and annotations to build the metamodel of the domain objects. This chapter explains what those conventions look like, and where annotations should be applied, when developing domain objects in Groovy.

Naked Objects allows validation and other business constraints to expressed either declaratively (using annotations) and/or imperatively (using supporting methods). For a given class member (property, collection or action) we can specify whether the member:

1. is visible, and if so, is it
2. is usable, and if modified, whether the proposed change
3. is valid

Or, slightly more pithily, *can you see it, can you use it, can you do it*.

Any public methods that don't represent properties or collections are interpreted as being actions. These are surfaced in the UI (as menu items or buttons) to provide arbitrary business behaviour (this is what makes *Naked Objects* applications more than a simple CRUD framework).

There are also a number of reserved method names that are used either as rendering hints or to define lifecycle callbacks.

Collectively these conventions and annotations define the *Naked Objects Programming Model*. This chapter should give you a good flavour of what it's like to writing applications for this programming model; but refer to the *Naked Objects* documentation for the complete reference.

The code fragments that follow are taken from the *Groovy Objects*' own test application (see the *Groovy Objects* subversion repository).

3.1. (Optional) Superclass

Typically domain objects subclass from `AbstractDomainObject` (in the *Naked Objects* applib, transitively referenced from *Groovy Objects*' own applib); for example:

```
class Claim extends AbstractDomainObject {
   ...
}
```

It isn't mandatory to subclass from `AbstractDomainObject`. All that *Naked Objects* requires is that it can inject its `DomainObjectContainer` into the domain object to support lazy loading and dirty tracking (the `resolve()` and `objectChanged()` methods, normally called by CgLib proxies). The `DomainObjectContainer` also allows your domain object to be able to instantiate and persist new instances (using the `newTransientInstance()` and `persist()` methods).

In practical terms, then, if you aren't able or don't want to subclass from `AbstractDomainObject`, just make sure you push down the above methods into your own objects (probably by way of a project-specific superclass).

3.2. Properties and Collections

Naked Objects follows the usual JavaBean conventions for properties, and so any Groovy property is picked up automatically by *Naked Objects*. This also works for collections (a JavaBean property that returns a `java.util.Collection`, `java.util.List` or `java.util.Set`).

For example, the `Claim` object is rendered like this:

The corresponding Groovy source code is:

```
class Claim extends AbstractDomainObject {

  boolean rush
  String description
  Date date
  String status
  Claimant claimant
  Approver approver
  List<ClaimItem> items = new ArrayList<ClaimItem>()

  ...
}
```

3.3. Title & Icon

Naked Objects uses the `title()` method to render a label for domain objects. For the `Claim` class, this is defined as:

```
class Claim extends AbstractDomainObject {

  ...

  String title() { status + " - " + date }

  ...
}
```

It's important that this is defined as returning a *java.lang.String*; a simple Groovy def is not sufficient.

In addition, you may want to define an `iconName()`. If present, this is used to locate the icon for the entity (meaning that different instances of the same type can render different icons). Otherwise, *Naked Objects* infers the icon from the class name. The icon is typically picked up from an images package (in `src/main/resources`).

3.4. Creating and Persisting Objects

Naked Objects will automatically inject any domain services into your domain objects, but to do this must know about then when they are instantiated. But it also requires to know about them so that it can track their persistence state. To do this, use the `DomainObjectContainer#newTransientInstance(Class)` method. Note that this also requires that your domain object has a `public` no-arg constructor.

Similarly, if you want to persist a domain object, use `DomainObjectContainer#persist()`.

If inheriting from `AbstractDomainObject`, then there are helper methods that delegate to the container for you.

3.5. Callbacks

There are a number of callback methods that *Naked Objects* will call on your domain object if present. One of these is `created()`, called after a transient instance is just instantiated. It's a good place to perform initialization logic (that would otherwise probably have lived in a constructor). For example:

```
class Claim extends AbstractDomainObject {

  ...

  void created() {
    status = "New"
    date = new Date()
  }

  ...
}
```

Note that the method must return `void` (Groovy's `def` returns a `java.lang.Object`, which is not what *Naked Objects* is looking for).

Other callback methods include:

- `loading()` and `loaded()`
- `persisting()` and `persisted()` (or `saving()` and `saved()` if you prefer)
- `updating()` and `updated()`
- `removing()` and `removed()` (or `deleting()` and `deleted()` if you prefer)

3.6. Annotations

Declarative business rules amount to applying annotations on the appropriate methods. But you should note that *Naked Objects* does not (currently) support annotations on fields, so it's necessary to put the annotation on the getter for the property.

For example, to indicate that a property is disabled (read-only), we can write:

```
class Claim ... {
    String status
    ...

    @Disabled
    String getStatus() { status }
}
```

Other annotations are used as hints for the user interface. For example the `@MemberOrder` is used to specify the order in which properties and collections appear in the UI:

```
class Claim ... {
    String status
    ...

    @MemberOrder("3")
    String getStatus() { status }
}
```

Another annotation is `@Named`, which commonly appears on action parameters if using built-in value types. For example, the `Claim`'s `addItem()` action looks like:

```
class Claim ... {

    void addItem(
            @Named("Days since")  int days,
            @Named("Amount")      double amount,
            @Named("Description") String description) {
        ClaimItem claimItem = newTransientInstance(ClaimItem.class)
        Date date = new Date()
        date = date.add(0, 0, days)
        claimItem.dateIncurred = date
        claimItem.description = description
        claimItem.amount = new Money(amount, "USD")
        persist(claimItem)
        addToItems(claimItem)
    }
    ...
}
```

The full list of annotations can be found in the *Naked Objects* applib, and are of course documented in the Naked Objects documentation.

3.7. Supporting Methods

Business rules can also be specified imperatively, using supporting methods. These methods are associated back to the class member (property, collection or action) using a simple prefix. For example, to imperatively disable the `addItem()` action for a `Claim`, we could use:

```
class Claim ... {

    void addItem(
            @Named("Days since")  int days,
            @Named("Amount")      double amount,
            @Named("Description") String description) {
       ...
    }
    String disableAddItem() {
        status == "Submitted" ? "Already submitted" : null
    }
    ...
}
```

Returning a non-null value means the action (or more generally class member) should be disabled; the string returned is the reason why the action cannot be invoked.

There are supporting methods for each of the three levels of business rules ("see it, use it, do it"), with the prefix being `hideXxx()`, `disableXxx()` and `validateXxx()`. The `hideXxx()` returns a `boolean`, the other two (as you've just seen) return a `String`. In the case of `validateXxx()`, the method takes arguments to allow validation to be performed, for example:

```
class Claim ... {

    ...
    String validateAddItem(int days, double amount, String description) {
        if (days <= 0) "Days must be positive value"
    }
    ...
}
```

There are a couple of other supporting methods that can be provided. The `defaultXxx()` prefix is used to provide a default either for a property of a newly instantiated object, or, more commonly, as the default for a parameter of an action. In the latter case the argument number is specified:

```
class Claim ... {

    ...
    int default0AddItem() { 1 }
    ...
}
```

In a similar vein, the `choicesXxx()` prefix provides a list of choices for a property or for an action parameter:

```
class Claim ... {

    ...
    List<String> choices2AddItem() { ["meal", "taxi", "plane", "train"] }
    ...
}
```

There's no requirement for `choicesXxx()` to tie in with `validateXxx()` or `defaultXxx()`, but they usually are consistent with each other.

Chapter 4

Writing Fixtures in Groovy

We can take advantage of one of Groovy's features to reduce the boilerplate when writing fixtures. This chapter explains how.

One of the classes provided by the Groovy runtime is `ObjectGraphBuilder`, which (as per its documentation) is "a builder for an arbitrary graph of beans that follow the JavaBean convention,... useful for creating test data for example". Which is, indeed, exactly what we need to do when we create fixtures for use with the in-memory object store.

The *Groovy Objects* applib extends this class by providing the `DomainObjectBuilder`, which additionally ensures that domain objects are instantiated through the `DomainObjectContainer`. The original `ObjectGraphBuilder` also needs to be told explicitly where the domain object packages are, so `DomainObjectBuilder` makes this easy to do in a typesafe way.

Let's see it in action. Here's the fixture from the *Groovy Objects* testapp, to create 3 `Employees`, some `Claims` and some `ClaimItems` within those `Claims`:

```
class ClaimsFixture extends AbstractFixture {

    @Override
    public void install() {
        def builder = new DomainObjectBuilder(getContainer(), Employee.class, Claim.class)

        builder.employee(id: 'fred', name:"Fred Smith")
        builder.employee(id: "tom", name: "Tom Brown") {
            approver( refId: 'fred')
        }
        builder.employee(name: "Sam Jones") {
            approver( refId: 'fred')
        }

        builder.claim(id: 'tom:1', date: days(-16), description: "Meeting with client") {
            claimant( refId: 'tom')
            claimItem( dateIncurred: days(-16), amount: money(38.50), description: "Lunch with
 client")
```

```
            claimItem( dateIncurred: days(-16), amount: money(16.50), description: "Euston -
 Mayfair (return)")
        }
        builder.claim(id: 'tom:2', date: days(-18), description: "Meeting in city office") {
            claimant( refId: 'tom')
            claimItem( dateIncurred: days(-18), amount: money(18.00), description: "Car
 parking")
            claimItem( dateIncurred: days(-18), amount: money(26.50), description: "Reading -
 London (return)")
        }
        builder.claim(id: 'fred:1', date: days(-14), description: "Meeting at clients") {
            claimant( refId: 'fred')
            claimItem( dateIncurred: days(-14), amount: money(18.00), description: "Car
 parking")
            claimItem( dateIncurred: days(-14), amount: money(26.50), description: "Reading -
 London (return)")
        }
    }

    private Date days(int days) {
        Date date = new Date();
        date = date.add(0,0, days);
        return date
    }

    private Money money(double amount) {
        return new Money(amount, "USD");
    }
}
```

The builder is instantiated by passing in the `DomainObjectContainer`, as well as one representative class from each package that holds entities to be built. In this case the `Employee.class` takes care of the employee package (for just `Employee` itself), while `Claim.class` represents the claims package (for both `Claim` and `ClaimItem`).

The DSL for building the object graph is just that defined by Groovy's `ObjectGraphBuilder`. To my eyes at least, this is easier to follow than its Java equivalent.

There is one limitation to be aware of, though, which relates to how the `Claim`/`ClaimItem` parent/child is wired up. It's important for the collection name in the parent (`Claim`) to match that of the class name of the child (`ClaimItem`), and the back reference in the child (if there is one) to match the class name of the parent. For the test app, this means that the collection in `Claim` is called `claimItems`. If this is irksome, then the `ObjectGraphBuilder` does define the ability to tweak its behaviour as to how the relationship name is inferred. (A future enhancement to *Groovy Objects* might be to solve this problem in the general case, by perform the wiring using the *Naked Objects* metamodel).