Helianto project

Reference Documentation

Maurício Fernandes de Castro

Helianto project : Reference Documentation

by Maurício Fernandes de Castro

Table of Contents

Preface

The Helianto Project was born after a discussion on how to leverage the competitive advantage of a small group of small organizations having different competences in the sotware business. The key value behind the Helianto mission development was to promote collaboration in disciplines where all members might have common requirements.

The name "Helianto" derives from the latin word for the sunflower plant. The idea showed up from an original sketch, where a couple of functional modules were drawn in a central corolla, surrounded by as many application "petals" as desired. The name still remains associated to the sunflower geometry, although the project focuses only the corolla.

Helianto is not meant to be a product to be used out of the box. Actually, it requires customization and some expertise in object oriented programming to become a productivity tool. It strongly relies on `Spring framework` to achieve the majority of its goals. Those who are familiar with Spring's concepts like dependency injection and aspect orientation will be more comfortable here. And those who are not, will find at Helianto a fast and convenient learning path.

At last, Helianto gained an inital momentum to become an open source project when the possibilities arround the "corolla" outnumbered the original group expectations. Since then, Helianto is distributed under the "Apache License v. 2.0".

Currently, Helianto has a work in progress status. This document is currently in the process of being written, so not all topics are covered. Contributions are welcome (including the language as we are not native speakers).

Chapter 1. The Helianto Project

Introduction

Helianto tries to capture common project issues and combine it with sound project pratices. Many designers need to model entities like customers or suppliers, payables and receivables, and so forth. Helianto provides extensible domain and service classes for them. Many designers have a growing interest on ORM. Helianto provides basic configuration for Spring and Hibernate to achieve this, as well as the corresponding OR mapping for the supplied domain classes. Many designers would refactor the code to reduce coupling. Helianto enforces the use of dependency injection. Some have realized EJB is too complex and will find here how to keep declarative transaction management simple. Some would like to benefit from a non invasive security framework. Helianto provides configuration and a template for a secure web application using Acegi Security.

Helianto strongly relies on Spring to achieve all of this. It also embeds a couple of design decisions that narrow the power of Spring. It is a trade off. In one hand, you have to follow a pre-configured application, but in the other, you may get ready shortly.

The key values to develop Helianto classes and resources is to keep wide scope and flexibility to ease composition or extension.

In a tipical use case, the Helianto user would:

- add the helianto*.jar packages to the application classpath,
- create the presentation layer to invoke the supplied service layer,
- customize the jdbc connection through the hibernate.properties file,
- test and deploy the appllication. .

If the application requirements become more specific, the user would also have to create or extend domain classes, map them to the persistence layer, create or extend service facades and wire them to the dependency injection container registry.

Throughout this document, many of the aspects of such customization will be described in deeper detail, along with some fundamentals of the collaborating open source packages.

Mission

Continuously develop an application base framework that:

- is as decoupled as possible from the presentation layer, allowing the desired customization to expose service and persistence layers that fits best its purpose,
- provides an extensible and flexible service layer to accomodate both simple and complex project requirements, either beeing partially implemented or extended through the customization,
- concurrently provides a common domain model to solve well known business problems and a persistence layer as decoupled as possible from the datastore, and
- enforces good programming practices like design patterns usage, rich documentation and extensive testing.

Acknowledgements

The Helianto project includes software developed by the Apache Software Foundation (http://www.apache.org), the Spring Framework Project (http://www.springframework.org) and the Acegi Security System for Spring Project (http://acegisecurity.sourceforge.net). It also includes software

from Hibernate (http://www.hibernate.org).

It is virtually impossible to say thank you to all the people who contributed in some way to the current state of this project. Many of them are project leaders of the packages mentioned above.

Alternatives

As stated before, the Helianto project is not meant to be used out of the box. This is primarily because the presentation layer is not part of the project mission. Other projects, like AppFuse (http://appfuse.dev.java.net) or Java Application Generator (http://jag.sourceforge.net) provide a more comprehensive support for a full architecture.

Software Requirements

The source code includes generics and enumerators, therefore a jdk 1.5 or higher is required.

Eclipse platform and the appropriate plugins as well as Apache Maven are recomended, although not required. Of course, there are many dependencies listed in the pom files, and Maven will help to have them ready to use.

Any sql database can be used as long as the appropriate properties are set in hibernate.properties at src/main/resources/.

How to Use this Guide

This guide has a mixed content. The sections "Architecture", "Dependency Injection", "Object relational mapping" and "Security" have a conceptual approach. All following sections describe the functionality of each module.

Architecture

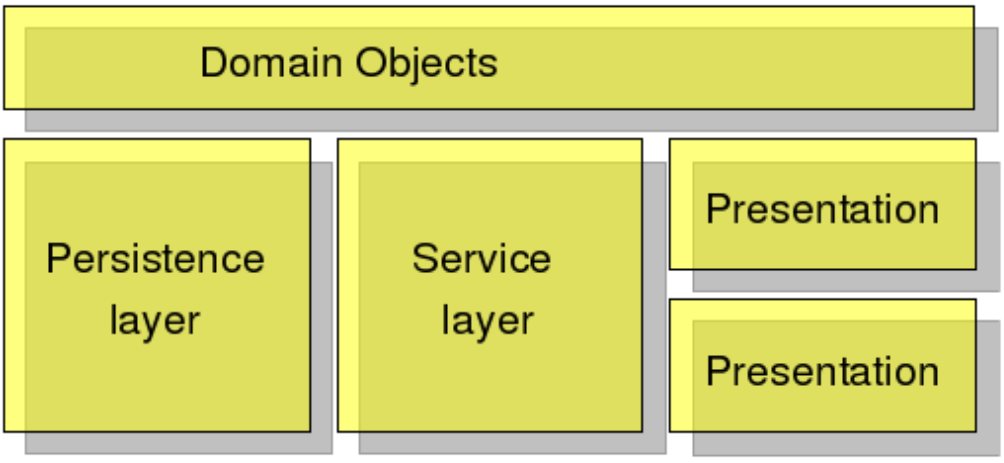


Figure 2: Architecture.

Dependency injection

First thing to say about dependency injection is that it is not invasive. It does not require your code to inherit state or behaviour from some class, or to implement any particular interface. It also means that you wil not have to take things away from your code if you decide you had enough from it, and will now follow the next buzzword framework.

The action may start just after two minor adjustments arround your code, or even, arround legacy code:

- at design time, you choose the objects to instantiate (and how they collaborate) in a declarative fashion,
- at run time, an IoC container takes first the control of the program flow, and uses the declared configuration to actually instantiate the objects.

The application logic may then follow, with or without additional collaboration from the container.

For those who are still asking what benefits a container to jumpstart the application may bring, let's say that:

- less coupling means less plumbing or glue code, so you are free to focus on the business,
- dependencies become explicit and away of the code, a good reason to follow the well known best pratice "program to an interface, not to the implementation",
- test driven development becomes easier, and
- dependency injection containers are lightweight.

Let's take an example from the Helianto configuration.

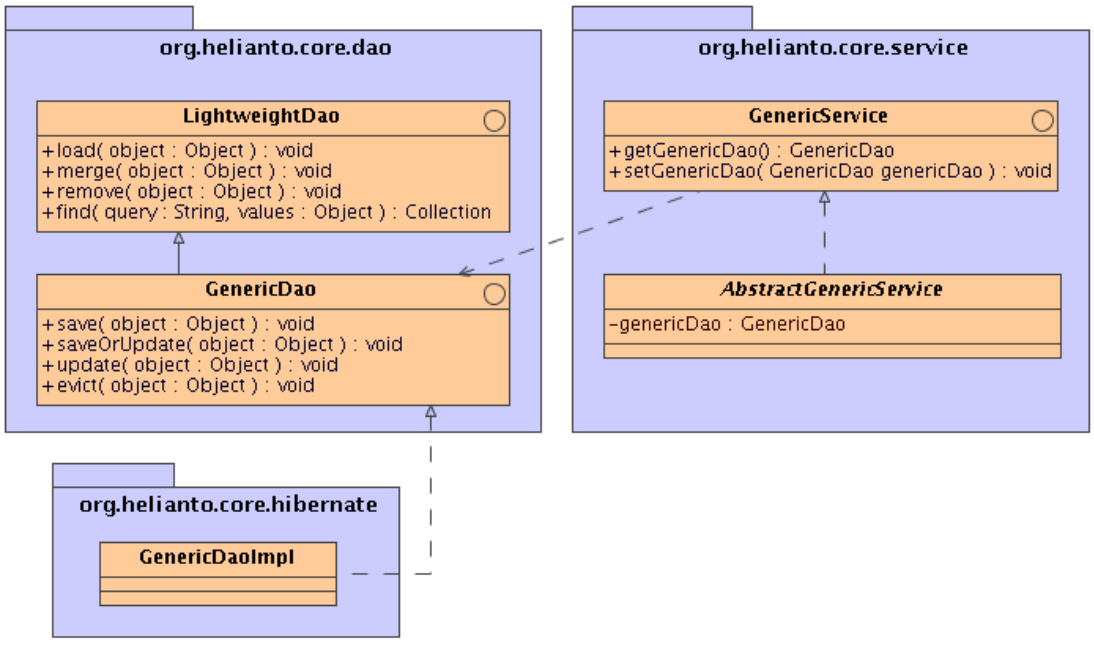


Figure 3: Dao's and dependencies.

The figure above shows the dependency between `AbstractGenericService` class and the inter-

face GenericDao. Any class descending from AbstractGenericService will, of course, have the same dependency.

At the time this document was beeing written, the current design choice was to implement a concrete class for the GenericDao interface using the Hibernate orm library (see the section about object relational mapping bellow). But if there is a new better orm package to be launched in the future, or if some Helianto user decides to implement a new concrete class for GenericDao based on Oracles's Top Link orm library, a change has to be done.

The approach suggested by dependency injection is to keep the service class dependency only on the GenericDao interface and delegate to the Spring container the responsibility to "inject" the concrete class, that implements the interface, through a setter method.

Here are two code snippets to illustrate this. The first (not recommended) shows a service object with dependencies to both the interface and the concrete class. The second one relies on a setter (and a not shown dependency injection assembler) to replace the dependency on the concrete class.

```
/**
 * Example 1: contains dependencies to GenericDao and GenericDaoImpl.
 */
public abstract class AbstractNotRecommendedGenericService
    implements GenericService {

    protected GenericDao genericDao = new GenericDaoImpl();
        ...
```

```
/**
 * Example 2: contains dependencies only to GenericDao.
 */
public abstract class AbstractGenericService
    implements GenericService {

    protected GenericDao genericDao;

    public void setGenericDao(GenericDao genericDao) {
        this.genericDao = genericDao;
    }
        ...
```

To use the Spring framework terminology, the application may instantiate either a BeanFactory, an ApplicationContext or WebApplicationContext to inject, via setter methods, a dependency at run time. The job of registering the instances to be instantiated and where they are injected is easily accomplished by a xml file. As an extra advantage, neither the injected beans nor the ones which receive the injection become dependant on any Spring framework code. Here is a snippet of a xml file.

```
<!--
                Hibernate generic dao implementation
```

```
-->
<bean id="genericDao"
        class="org.helianto.core.hibernate.GenericDaoImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
...
<!--
                Core service bean
-->
...
                <bean id="coreTarget"
                        class="org.helianto.core.service.CoreMgrImpl">
                        <property name="genericDao" ref="genericDao"/>
                </bean>
...
```

The beans in the registry can be refereced by their ids.

Now, if the programmer writes a new implementation for `GenericDao`, besides the new code, the `CoreMgrImpl` would remain untouched and the new file might contain:

```
<!--
                Toplink generic dao implementation
-->
<bean id="genericDao"
        class="org.helianto.core.toplink.GenericDaoImpl">
    ...
</bean>
...
<!--
                Core service bean
-->
...
                <bean id="coreTarget"
                        class="org.helianto.core.service.CoreMgrImpl">
                        <property name="genericDao" ref="genericDao"/>
                </bean>
...
```

In other words, dependecies are managed in a declarative, not programatic way.

That summarizes why dependency injection became a fundamental block for the construction of the Helianto code. The `Spring framework` dependency injection container was the preferred implementation, not only because of its stability or consistency, but also because it offers many other advantages:

- easy integration with orm libraries,
- declarative transaction management,
- allows for aspect orientation, and as an immediate consequence, for strong but not invasive security management (see bellow),
- easy mvc integration, and also the new `Webflow` package.

Object relational mapping

[pending]

Security

The helianto project relies on the `Acegi-security for Spring` package to provide user authentication and authorization. The following sections have a restricted view of some main objects and concepts, while the Acegi web site documentation is comprehensive and recommended for those who look for deeper understanding.

Security object

The security object plays a major role inside `Acegi-security for Spring` package as it coordinates the transitions required to secure an object. For the purposes of this document, the `FilterInvocation` security object is examined, and how it enables HTTP resources to be secured.

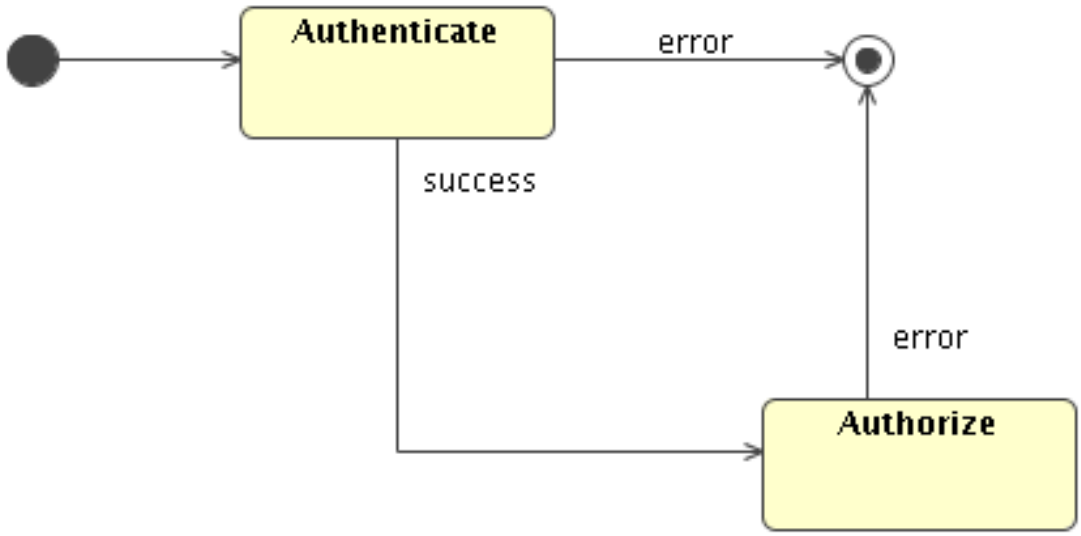


Figure 1: The security transitions.

The security interceptor has some important collaborators with distinct responsibilities:

- An `Authentication` object to hold the username, password and the authorities granted to the user.
- A `ContextHolder` which holds the `Authentication` object in a `ThreadLocal` -bound object.
- An `AuthenticationManager` to authenticate the `Authentication` object presented via the `ContextHolder`.

Core functionality

The `helianto-core` package

The `helianto-core` package plays the major role for Helianto funcionality. It comprehends many

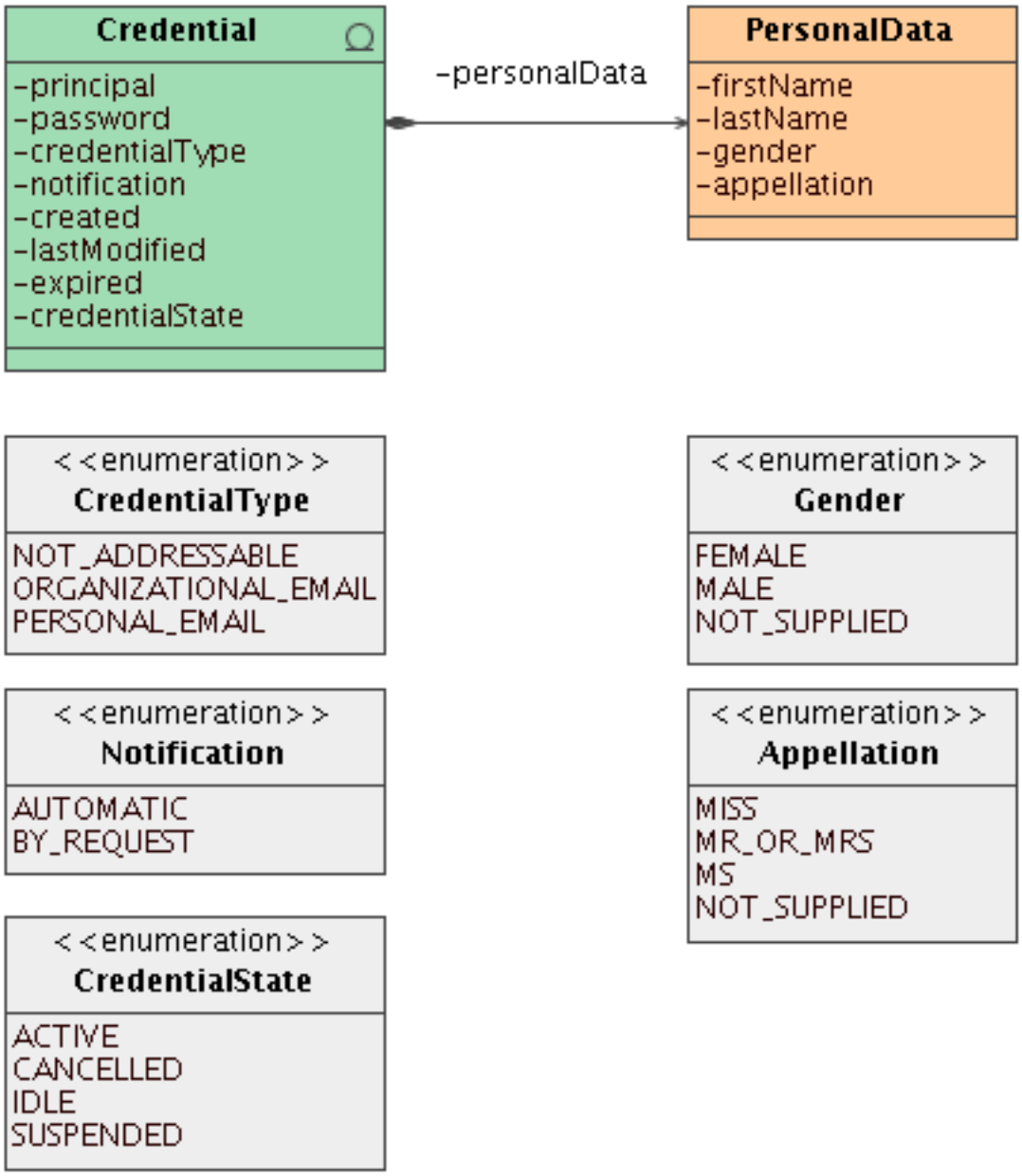
responsibilities:

- support basic Spring, Hibernate and Acegi configuration, wired togheter to provide common functionality to all other modules,
- fully implement the persistence layer, exposing a standardized dao interface,
- provide for multiple entity resolution, as entities may be organizations, personal owners, customers, suppliers, and so forth - content of one entity should be visible to others under controlled circumstances, if an internet distributed application is desirable,
- implement user identity, taking into account that a user may share privileges among different entities,
- allow for hierarchical authorization control, follwing the chain: supervisor -> administrator -> moderator -> user, and provide for standard event notification.

The Credential domain object

A `Credential` provides the security system with authentication information.

A `Credential` is usually a person, but it may represent also a non-personal identity. A `Credential` is required for a person to be recognized by the system. The core domain model is designed to have a `Credential` playing different roles, even for different entities.



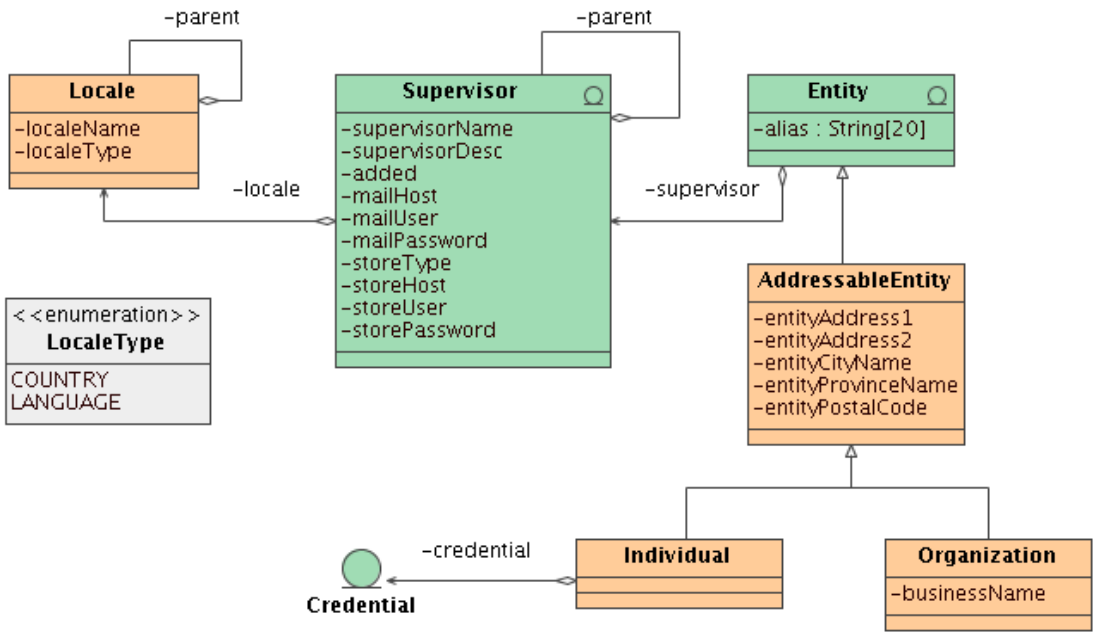
Credential and personal data.

A `Credential` must have a principal to be uniquely identified. The preferred way to supply a `Credential` with a globally unique principal is the e-mail address. A non-addressable principal may be used if the `Credential` is related to a parent having an addressable principal, or in a chain, up to at least one addressable principal.

The Entity domain object

Helianto core functionality relies on the `Entity` abstraction. It represents the root for several hierarchies, for example, all users or all processes of an `Entity`. It may represent either an organization or part of an organization, that is in control of such hierarchies.

The `Entity` has a simple interface to allow easy extension. Its primary responsibility is to grant read-write access to the `alias : String [20]` property, which is implemented as unique in a datastore.

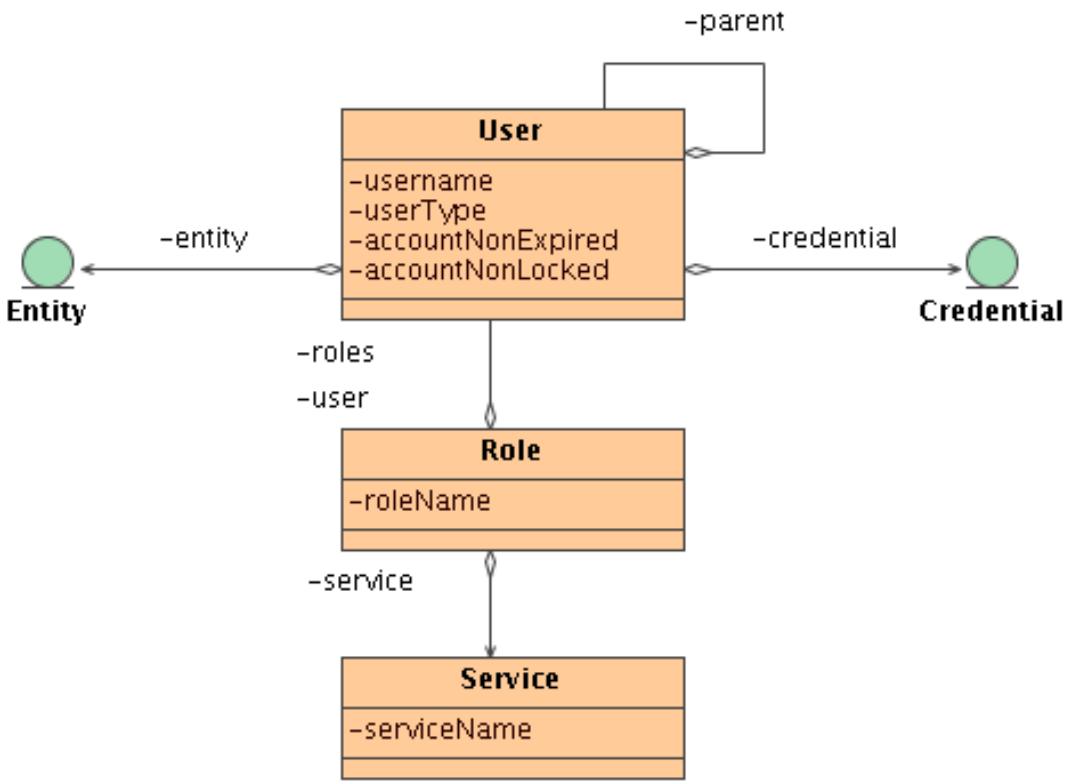


Entity model.

Each `Entity` must recognize an `Supervisor` property, specially if an abstraction to keep an `Entity` collection as a territory is required.

The User domain object

A `User` represents the relationship between a `Credential` and an `Entity`. It is the mechanism used by a person, or any other identity, to have granted authorization to play a `Role` within an `Entity`.



User model.

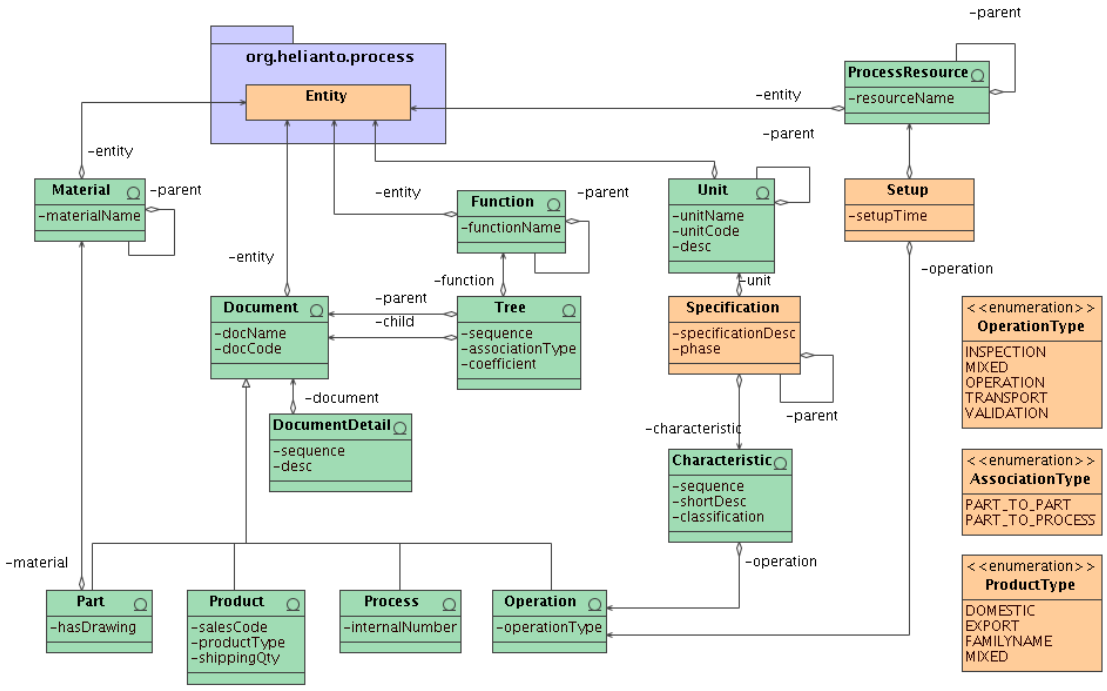
Process functionality

The `helianto-process` package

The `helianto-process` package, as the name suggests, introduces a comprehensive process approach to Helianto. Processes may be taken as organizational or production, and may be related to parts, products or other processes. The module responsibilities are:

- provide a base class, `Document`, to be root for all process components, allowing for polimorphism among parts, products, processes and operations,
- provide product and process trees, and services to maintain them,
- provide manufacturing specific domain and services,
- provide base classes for resource management.

The Process domain model



Last updated 20051010.

[Pending]

Sales functionality

The `helianto-sales` package

The `helianto-sales` package is designed to provide price information to products. The module responsibilities are:

- handle, `Proposal`s and `Price` list,
- provide services to calculate prices.

The Sales domain model

[Pending]

Finance and accountability functionality

[pending]

A template application including access control

[pending]