IGLU

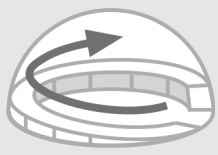
MANUAL FOR THE IGLU CONFIGURATION FRAMEWORK

1 CONTENTS

2 INTRODUCTION

Iglu is an atomic framework, written in Java. It's a reference implementation for research focused on the configuration of an application in terms of modules, clusters, layers and their relations. It aims to help developers organize high-level separation of concerns and adhere to their application design. Iglu was developed by Jeroen Meetsma, partner at IJsberg Software Architects.

Iglu Configuration is an atomic framework. Atomic frameworks are by definition the smallest possible frameworks to address a particular problem domain. Iglu Configuration addresses high-level partitioning of an application in subsystems, layers and modules. It provides a few mechanisms to structure relations between loosely coupled components: Inversion of Control (IoC), the proxy pattern, Interface Invocation Interception (a subset of AOP) and the publish-subscribe pattern. All of these mechanisms have been available as patterns or as part of various other frameworks. By combining these mechanisms in a single dedicated framework, Iglu Configuration aims to close the gap between the application design from a configuration and administration perspective and the actual implementation.



About the Igloo

An igloo is made out of large, light and simple building blocks. An igloo is constructed in a spiral. It reflects the way an engineer designs a construction by continuously extending and adjusting a design while judging the result from an ever shifting angle.

2.1 BACKGROUND

In addition to object orientation (OO), a number of concepts have emerged that have given us more power to structure our software and to create more substantial structures than just classes. They are all more or less applied in Iglu Configuration.

The most notable programming concept is the notion of Design Patterns itself. By simply giving useful object oriented constructs a name, developers could now recognize and reuse these concepts. Patterns allow you to get rid of unnecessary dependencies; every pattern is a miniature design to separate concerns in yet another way for yet another situation. The patterns used in Iglu Configuration are Facade, Proxy, Publish-Subscribe and Inversion of Control.

Aspect Oriented Programming (AOP) is a concept that liberates OO-developers from having to address different concerns in one model. It allows OO-developers to create pure domain models and move cross-cutting concerns, like security or monitoring, to different area's.

Another concept is Component Based Development (CBD). Classes are too small and often too entangled to function as components that can be used to assemble applications. So the concept of "components" emerged: coarse-grained, independent pieces of software that could be assembled like Lego blocks. Iglu is more or less based on this idea. Although he flexibility of ordinary Lego is in fact very limited, the concept is pretty feasible for the technical infrastructure or the administrator's view of an application. The Lego metaphore is however totally inappropriate for modelling the majority of problem domains. The complexity of domain-specific situations must be reflected in a proper object oriented models. Otherwise it pops up in endless blocks of if-then-else code.

Iglu Configuration serves as an invisible exoskeleton that embeds the application's functionality. This exoskeleton is assembled out of Modules and Clusters of Modules. Because an application contains code that can not simply be assembled using a simple (declarative) connection pattern, the modules proposed in Iglu must be seen as gateways for constructs that are as complex as the domain requires.

The actual configuration of an application should be located in one place and should match the design. Different test configurations supporting unit testing and TDD. An assembled structure can be made queryable at runtime by administrators.

3 EMBEDDING

Some classes in an application have a structural meaning; they perform a particular task or exist as access point for complex structures. Instances of such classes, often singletons, can be embedded in objects which represent bulding blocks. By doing so, the structure can be formalized and used to enforce a particular design.

Useful building blocks are typically technical or domain-specific services, represented by a single class that acts as facade or adapter or as a service in its own right. Instances of such classes are embedded in Modules, the equivalent of Java Beans in other frameworks. Iglu Configuration comes with a standard implementation of a module that simply takes the embedded service as an argument:

```
Shop drugstore = new ShopImpl("The Drugstore");
Module shopModule = new StandardModule(drugstore);
```

A module both facilitates and administers relations to other modules. It shields the embedded object from direct access. The module formalizes the way the service can be accessed: it's only possible to access implemented interfaces by proxy like this:

```
Shop shop = (Shop)shopModule.getProxy(Shop.class);
```

It's also possible to retrieve all implemented interfaces:

```
Class[] implementedInterfaces = shopModule.getInterfaces();
```

3.1 PROPERTY INJECTION

Administration of application settings is facilitated by a Modules ability to perform setter injection.

A service wrapped in a Module can be configured using a property bundle:

```
Properties shopProperties = new Properties();
shopProperties.setProperty("ShopCode", "ET-27");
shopProperties.setProperty("MaxOrdersDeliverablePerDay",
"2000");
shopProperties.setProperty("MinOrderSize", "10");

shopModule.setProperties(shopProperties);
```

StandardModule supports setter injection to set the properties in the wrapped object. It will try to convert strings to the appropriate primitives or objects if setters require such arguments. In addition to this, StandarModule will also invoke a method `setProperties(java.util.Properties)` on the wrapped object if such a method is available. This can be used as an alternative for setter injection, but it may also be used as a trigger that properties have been changed and (renewed) initialization is required.

3.2 INTERFACE INVOCATION INTERCEPTION

Module interfaces often represent boundaries of subsystems. This makes their invocations eligable as monitoring indexes or checkpoints.

Iglu Configuration enables developers to intercept calls to interfaces by allowing to add plain InvocationHandlers like the following one to modules:

```
public class ProductInquiryCounter implements
InvocationHandler {

     int countedInquiries;

     public Object invoke( Object proxy, Method method,
                           Object[] parameters) throws
                      Throwable {


     if(equals(method.getName().startsWith("findProduct"))) {
               countedInquiries++;
          }
          return method.invoke(proxy, parameters);
     }

     public int getNrofInquiries() {
          return countedInquiries;
     }
```

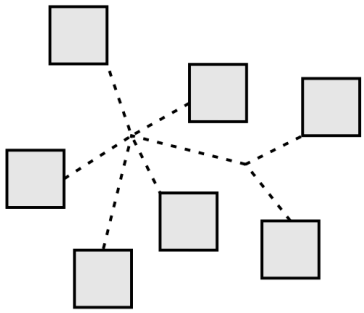
```
}
```

The code to add the invocation interceptor looks like this:

```
ProductInquiryCounter productInquiryCounter =
                              new ProductInquiryCounter();
shopModule.setInvocationInterceptor(
                              Shop.class, productInquiryCounter);
```

4 CLUSTERING

Modules can not reference each other unless they are part of a cluster. This is done to promote the formation of substantial subsystems or layers, corresponding with the design.



A Module is added to a Cluster like this:

```
Cluster cluster = new StandardCluster();
cluster.connect("Drugstore", shopModule);
```

The modules that form a cluster relate to each other. They share the same abstraction level or serve a common purpose. Each module is aware of its place and task in a cluster and it knows on which other modules it depends. Connections within the cluster are trusted. Modules are registered in a Cluster by some ID. It is not possible to register two modules by the same ID. It is however permitted that a module is registered by several ID's.

A Map containing all modules of a cluster can be obtained by invoking `getInternalModules()`. The following, rather elaborate statement gives us a reference to shop proxy.

```
Shop shop = (Shop)cluster.getInternalModules().
                        get("Drugstore").getProxy(Shop.class);
```

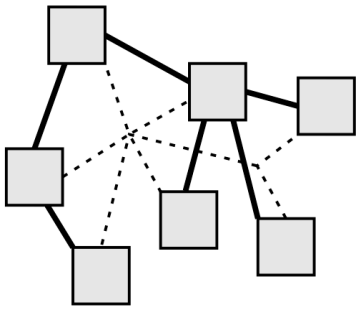
4.1 REFERENCE INJECTION

It is not necessary to obtain proxies by the `getInternalModules` method. Suppose our shop is a facade for a number of services, including a photo print service. Our Shop implementation now depends on a PhotoPrintService implementation. If ShopImpl has a method `setPhotoPrintService(PhotoPrintService photoPrintService)`, a reference to a PhotoPrintService will automatically be set on connection of such a module:

```
cluster.connect("PhotoPrintService", photoPrintServiceModule);
```

This mechanism works the other way around as well. If the Shop would be connected to the cluster after connection of the PhotoPrintService, the reference would be injected upon

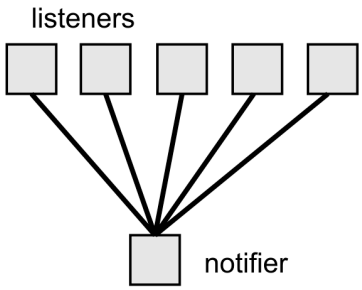
connection of the Shop. In fact a Cluster will resolve any possible reference, as expressed by the presence of appropriate setters.



This mechanism is also known as Inversion of Control, since it's the configuration that controls creation of objects and references instead the objects themselves.

4.2 REFERENCE REGISTRATION

Not all connections between modules have to be well defined one-to-one relationships. A module that acts as registry will want collect all references of a particular type. Registered modules may be interested in getting notified if some event occurs. In another case a module may want to use any module that exposes a certain interface, regardless of its ID.



These situations can be addressed using a publish-subsribe pattern.

If an object embedded in a module contains a method named "register" that declares one particular interface as argument, this method will be invoked for each and every module in a cluster that exposes this interface. Here's a class that defines a register method:

```
public class ShoppingCenter {
      private HashMap<String, Shop> listedShops =
                                    new HashMap<String, Shop>();

      public void register(Shop shop) {
            listedShops.put(shop.getName(), shop);
      }
}
```

By adding a ShoppingCenter Module to the cluster, the previously connected shop will now be registered with the ShoppingCenter:

```
ShoppingCenterImpl shoppingCenter = new ShoppingCenterImpl();
Module shoppingCenterModule = new
StandardModule(shoppingCenter);
cluster.connect("Shopping Center", shoppingCenterModule);
```

Reference registration is indispensible for outbound communication and registry functions.

5 LAYERING

Layering is a design concept that describes the separation of application parts in different abstraction levels. It is mandatory that layers are stacked. An object in one layer may reference objects in its own layer and objects in an adjacent layer. Layering is seldom addressed by component frameworks. Classes that belong to a certain layer are usually packaged together and sometimes maintained as a self-contained unit. To keep an implementation in line with the design, development must normally be guided and reviewed.

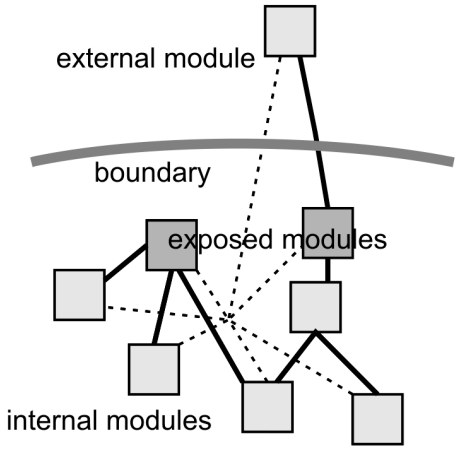
Iglu Configuration already implies the grouping of related Modules in Clusters. Clusters are also equipped to represent layers in an application. If the cluster represents itself as layer, it creates a ***boundary*** that restricts access to its contents. Access to modules in a cluster from a module outside the cluster is still possible. Let's create a new cluster and add some modules to it:

```
Cluster cluster = new StandardCluster();
cluster.connect("PhotoPrintService", photoPrintServiceModule);
cluster.connect("Drugstore", shopModule, Shop.class);
```

The shop module is now connected using a method that takes as a third argument single class or an array of classes that represent interfaces to be exposed to modules outside of the layer. A Layer can be obtained as follows:

```
Layer serviceLayer = cluster.asLayer();
```

Now from outside the layer it is not possible to obtain a proxy to PhotoPrintService. The call `serviceLayer.getProxy("PhotoPrintService", PhotoPrintService.class)` results in a `ConfigurationException`.



A reference to Shop, which is exposed by module "Drugstore", can be obtained:

```
Shop shop = (Shop)serviceLayer.getProxy("Drugstore",
Shop.class);
```

5.1 REFERENCE INJECTION AND REFERENCE REGISTRATION

Connections to modules outside of the cluster, passing the boundary between layers, are untrusted. The connection process differs from the process followed for connections within a cluster. Suppose we have a BasketImpl class that declares the following method:

```
public void setDrugstore(Shop shop) {
      this.drugstore = shop;
}
```

The method will be invoked and the referrence set on connection of the basket to the cluster via the serviceLayer:

```
BasketImpl basket = new BasketImpl();
Module basketModule = new StandardModule(basket);
serviceLayer.connect(basketModule);
```

The basket is connected to the cluster as external module. When connecting via a layer as external module, the `connect` method does not permit a module ID to be passed. The reason is that a lower layer does not depend on a higher layer and is not supposed to have knowledge of modules in the higher layer. Consequently references to external modules are not set in internal modules.

Reference Registration works for external modules and internal modules that act as registry. Consider a class CustomerSurvey which declares the following method:

```
public void register(Basket basket) {
      registeredBaskets.add(basket);
}
```

It can be connected to the cluster as internal module like this:

```
CustomerSurvey customerSurvey = new CustomerSurvey();
cluster.connect("customer survey",
                      new StandardModule(customerSurvey));
```

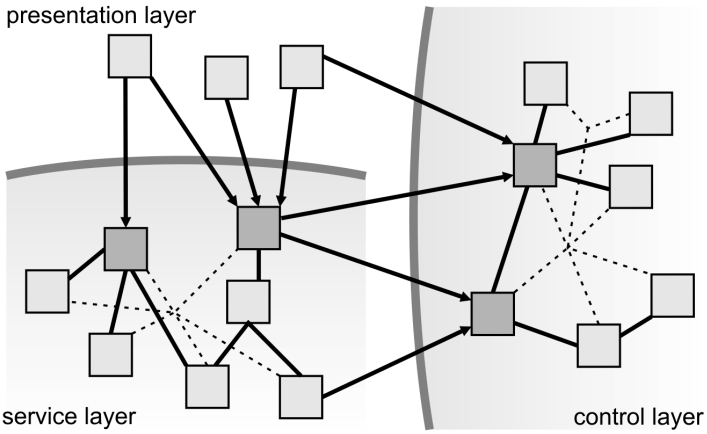
The previously connected basket will be registered in CustomerSurvey. This does not work the other way around! An external module can not register internal modules.

The Iglu layering solution frees developers working on another abstraction level from having to consider too many options; they can concentrate on the functions that are exposed.

6 USE OF IGLU

Iglu is the distilled remainder of a number of years of experience with modularisation and pluggability. It contains the essential elements for component based development in its purest form. Iglu is meant to serve as the acid test for a theoratical approach to the partioning of software. A paper is currently being finalized; a forerunner (that gets stuck on practical implications) can be found on the web:
http://ijsberg.org/documents/PESA_two_dimensional_layering.pdf.

A practical and in most cases sufficient layering structure consists of only two formal layers, a horizontal presentation layer and a vertical control layer.



Iglu has currently no practical use, other than being an alternative dependency injection framework. It will be extended with a number of modules that facilitate development of standalone server software.