The Islands Implementation
Reverse Engineering with an evaluation of its suitability as a BabyUML Foundation

Trygve Reenskaug, *Department of Informatics, University of Oslo*
trygver<at>ifi.uio.no
http://heim.ifi.uio.no/~trygver

Abstract

This article reports a reverse engineering study if the Islands implementation. It focuses on the object structure of a simple island in its environment and the process of creating a new island with two member objects. The result was a surprise to me and might be of interest to serious users and implementors of Islands and Tweak packages.

Superficially, the Islands implementation seems to realize concepts that are similar to the BabyUML Component. The purpose of this study was to establish if a BabyUML implementation should be based on Islands. The regrettable conclusion was negative; the details of the Islands implementation were at odds with the architecture of the BabyUML component.

I want increased confidence in my programs. I want my own and other people's programs to be more readable. I want a new discipline of programming that augments my thought processes. I create and explore a new discipline in my BabyUML project. I select, simplify and twist UML and other constructs to demonstrate how they help bridge the gap between me as a programmer and the objects running in my computer The key is to let my code explicitly specify three important aspects of my programs: What are the objects, how are they linked, and how do they collaborate to realize their goals. The focus is on the run time objects; the classes that specify them is moved towards the background.

I gave an overview of three promising coding constructs in an earlier article[1]. A fundamental BabyUML idea is to divide the objects space into an hierarchical structure of *components.* A BabyUML component looks like a single object when seen from the outside and is characterized by its provided and required interfaces. Inside, the component contains a system of interacting member objects that collectively implement the provided interfaces and use the required ones.

A Squeak construct called *Islands* looks very promising as a foundation for the BabyUML Component:

> *An island is a unit of isolation for sets of interacting objects. Any object lives on precisely one island and never "leaves" its island. Put differently, no two objects on different islands can refer to each other directly. Instead, objects on different islands refer to each other via so-called "far references" (FarRef) which are well-known objects controlling and relaying the incoming messages to its designated receiver on the correct island.*[2]

I will first reverse engineer a simple Islands example in order to better understand its fundamental mechanisms. I will then consider if and how these mechanisms can be support the BabyUML component.

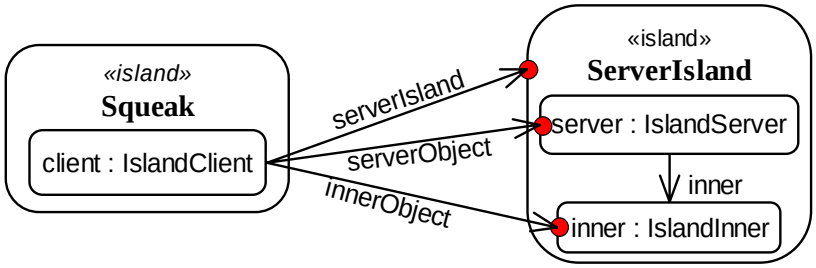
---

1. Reenskaug: *Towards A New Discipline of Programming*. Oslo, 2005-09-16. http://heim.ifi.uio.no/~trygver/2005/babyuml/newdiscipline.pdf
2. *???: Islands*. http://tweak.impara.de/TECHNOLOGY/Whitepapers/Islands/

THE *ISLANDS* MECHANISM.

Figure 1 shows a simple example with two islands. The first island is Squeak, it has one interesting object, client, an instance of IslandClient. The second island is ServerIsland. It contains two objects, server and inner, instances of the IslandServer and IslandInner classes respectively.

Figure 1: A simple example with two islands.



Access to the objects within the ServerIsland shall be protected by the FarRef objects marked as small, red circles in figure 1.

Example code

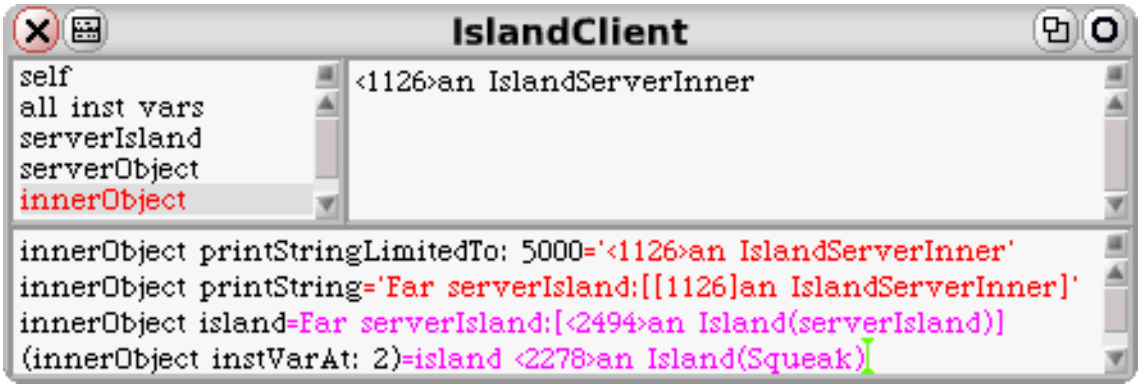
Figure 2: The example is coded in three classes. :

```
Object subclass: #IslandClient
        instanceVariableNames: 'serverIsland
    serverObject innerObject'
        ...
IslandClient>>test
        serverIsland := Island named: 'serverIsland'.
        serverObject := serverIsland new: IslandServer.
        innerObject := serverObject runServer.
        self inspect.

Object subclass: #IslandServer
        instanceVariableNames: 'inner'
        ....
IslandServer>>initialize
        super initialize.
        inner := IslandServerInner new.
IslandServer>>runServer
        ^inner
Object subclass: #IslandServerInner
        instanceVariableNames: ''
```

Figure 2 shows the example code. (A large number of trace statements have been omitted). I run IslandClient new test and get an inspector on the client object. The inspector is shown in figure 3.

Figure 3: The inspector opened by running IslandClient>>test.



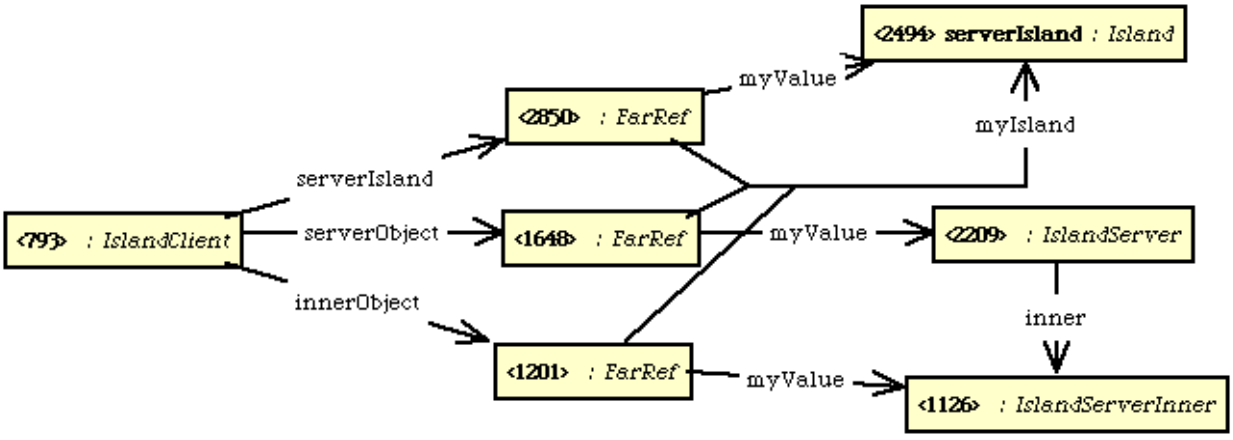
The above inspector on the client object is an illusion because it shows the innerObject as being an IslandServerInner instance with oop=1126. The FarRef object is invisible because it is transparent to almost all messages.

A closer study of the Inspector code explains this phenomenon. The Inspector uses printStringLimitedTo: 5000 to present the innerObject value. The FarRef>>doesNotUnderstand: method forwards this message to the <1126>inner object. In contrast, printOn: and thus printString *is* implemented in FarRef and is handled there. The two first *print it*-statements in figure 3 illustrates this difference. (Note: The printOn: method has been modified to show the object's <oop>.)

I hacked the BabySRE reverse engineering tool to handle this object structure. The essential objects are shown in figure 4. This is as expected and conforms to our goal shown in figure 1.

Note that the island objects do not know their island; while all FarRef objects *do* know it.

Figure 4: Essential experiment objects.



The creation of the serverIsland

I created the serverIsland in IslandClient>>test with the statement serverIsland := Island named: 'serverIsland'. The code is

```
Island>>named: aString ^self new name: aString.
```

This looks like a perfectly normal instantiation, yet the code does not return an instance of Island, but an instance of FarRef?!?

The secret is in self new. The code is:

```
Island class>new
        | new ref |
        new := self basicNew initialize.
        ref := new asFarRef: new.
        ^ref valueOn: self island
```

So new returns a FarRef to the new instance. The FarRef is transparent to name:, but it is the FarRef that is returned from the named: method.

The creation of the serverObject

The Island documentation says that I create a new instance as a member of an island by a message to that island: serverObject := serverIsland new: IslandServer. The result of this statement makes serverObject point indirectly to the new instance through a FarRef as shown above. But the code for Island>>new: is misleadingly simple:

```
Island>>new: aClass
        "Create an instance of aClass"
        ^aClass new
```

Again, we see that server object is returned; yet I get a FarRef to it. The trick is that while a FarRef is transparent to message sends, it is far from transparent to the returned values. The code that inserts the FarRef in front of the returned value is very complex. Figure 7 on page 6 is a Message Sequence Chart showing a simplified trace of the process. It shows that the <1648>FarRef instance is created in the asFarRef:-method in the <2494>serverIsland object. The FarRef initially points to nil. It gets its values deep down in exception blocks in the dictionary-like exports : FarRefMap object. (The relevant method is highlighted red in the chart.)

An island is a boundary

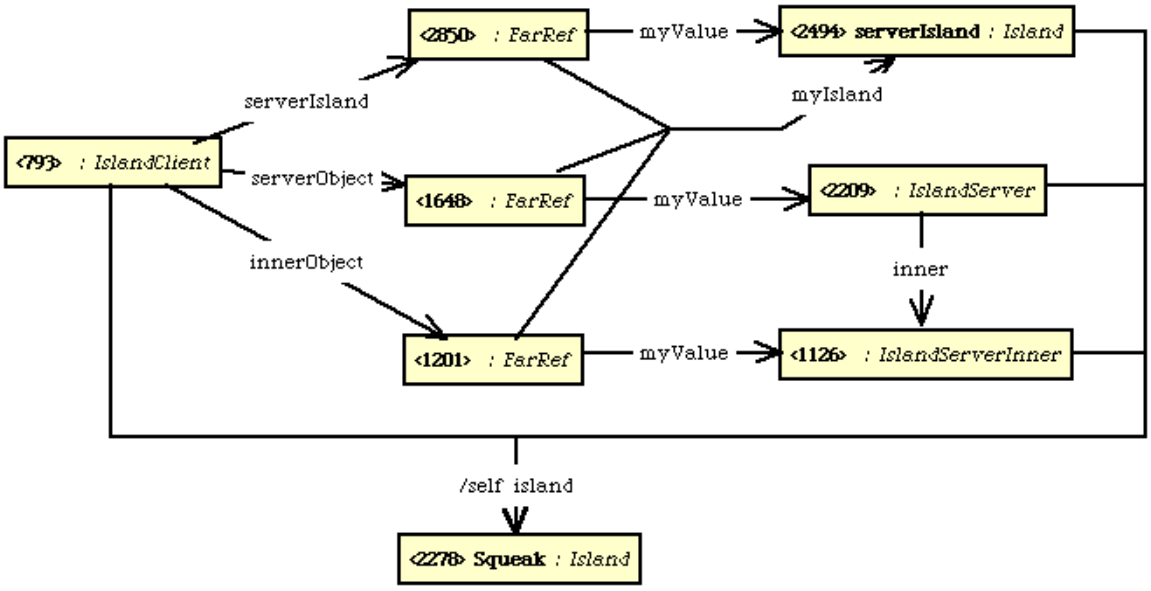
It is worth noting that figure 1 on page 2 is misleading in that an island is an illusion created by the FarRef accessors. The objects themselves are floating freely and their island property depends on the island of the observer. The two last print it statements in figure 3 on page 2 show that the same object appear to reside on two different islands simultaneously.

The explanation is in this code:

```
island
    "Answer the receiver's island"
    ^Processor activeIsland
```

The receivers island is not a property of the receiver, but on the active process. So the receiver appears to belong to the island of whoever asks the question. This point is also brought out in the object diagram of figure 5 where the serverIsland objects believe hey are on the Sqeak island because they were asked from that island.

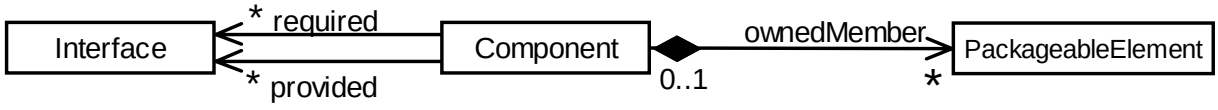
Figure 5: The the serverIsland objects' answer to self island.



ISLANDS AS A FOUNDATION FOR BABYUML COMPONENTS

A BabyUML *component* is an object that is characterized by its provided and required interfaces and that hides an internal structure of member objects. It is adapted from the UML component metamodel shown in figure 6. We see that the member elements are a composite aggregation, i.e., they are deleted if the component itself is deleted.

Figure 6: Snippet of the UML 2.0 metamodel.



The *Islands* package appears to be fundamentally different. It is a boundary around a set of free floating objects; there is no record of the identity of these objects. Indeed, an object can apparently be on several islands simultaneously.

UML also has the concept of *Ports* that specify distinct interaction points between a classifier such as a component and its environment. At a first glance, the UML *Port* and the Islands' *FarRef* seem similar. This study has reveled that their semantics is totally different and that the complexity of the FarRef makes it unsuitable as a simple access point.

CONCLUSION

Superficially, the *Islands* package seems to realize concepts that are similar to the UML Component. This study reveals fundamental differences. The FarRef class is too complex to form the superclass of a simple BabyPort class of access points. The Islands class could possibly be extended to a BabyComponent class, but the dynamics of the *Islands* object creation and component membership with the main logic in the FarRef is contrary to the idea of a BabyComponent that encapsulates and manages its member objects.

The conclusion is, regrettably, that Islands should not form the foundation for the BabyUML component.

Figure 7: Instantiation of the server object from outside its island