Generation of Graphical User Interface by Metaprogramming Application to Individual Based Models with Java Reflection API and Annotations

Nicolas Dumoulin[*], Thierry Faure[*], David Hill[**]
Nicolas.Dumoulin@cemagref.fr
[*] Cemagref, Laboratoire d'Ingénierie pour les Systèmes Complexes
24 avenue des Landais, BP 50085 63172 Aubière Cedex, France
[**] LIMOS – UMR CNRS 6158
Campus des Cézeaux, BP 125, 63172 Aubière Cedex, France

Keywords : metaprogramming, reflection, metamodelling, Java

Abstract

The introduction of a new reflection Application Programmer's Interface in the last version of the Java development platform enables the production of advanced programs known as metaprograms. This paper describes how new features such as annotations can be used to produce more flexible code. A revised version of the Observer pattern is proposed and simplifies the addition of an observable subject. Annotations are also used to generate automatically a Graphical User Interface at runtime. This programming technique is then applied to an Individual Based Model dealing with freshwater fishes.

1 INTRODUCTION

Reflective programming can be used to develop more flexible programs. To improve the support of reflection, general-purpose annotations (known as metadata) and introspection are introduced in modern languages like Java and development platforms like .Net. In this paper, we expose how such features enhance the semantics of a source code and facilitate the modification of runtime behavior. We first illustrate this with an implementation of the Observer design pattern. In this application, annotations are used to describe the role of an observable object. A second example is given aiming at the automatic generation of a graphical inspector allowing the edition of object properties. In this example, annotations are used to add attributes to the graphical layer, they are also used to specify which method should be invoked to display an object. The last part of this paper underlines the benefits of reflective programming when used to build a software framework for individual-based modeling. The software framework achieved with such programming features improved considerably the development and the exploitation of an individual-based freshwater fishes model.

2 BACKGROUND ON REFLECTION AND METAPROGRAMMING

In computer science, reflection is mainly the ability of a program to observe and possibly modify dynamically its structure at runtime in languages that make a difference between compile time and execution time. The reflection could also be applied to variants of this concept, they will not be discussed here; the interested reader will find more information in Wikipedia (reflection). Within the context of our first sentence, reflection supposes that information about the program structure is stored in the executable code. This reflection feature is provided by "metadata", present at the source code level and also in the executable code. In addition, this also supposes that the language provides a library or intrinsic features to discover and modify source code, to convert and assign at runtime a string into a reference to a first-class object (class, method…) and to evaluate a string at runtime as if it were a source code statement. The java language enables reflection using the Java package *java.lang.reflect*, this package provides classes and interfaces for obtaining reflective information about classes and objects. The code below shows the flexibility at runtime, first we find a class named *ReflectExample*, then the code takes a reference to the method named *methodExample* and launches the execution of this method with a newly created instance of the class selected at runtime.

```
Class  aClassRef= Class.forName("ReflectExample");
Method aMethod  =
       aClassRef.getMethod("methodExample",null);

aMethod.invoke(aClassRef.newInstance(), null);
```

More flexibility is provided to the developer since the code is not "hard-coded". The main drawback being that this code is not extremely safe since in Java the compilation phase was not able to check its final meaning (syntax and semantic). At runtime, if the class *ReflectExample* does not exist a "runtime error" will occur, whereas a hard-coded version using the name *ReflectExample* would have generated a compile-time error.

3 METAPROGRAMMING WITH THE JAVA REFLECTION API

3.1 Presentation of source code annotations

The last release of the Java Virtual Machine (JVM) of Sun Microsystem (known as release 5.0) introduced general-purpose annotations known as metadata. This gives to developers the ability to annotate several java elements such as classes, attributes, methods and annotations themselves. Furthermore, we have the possibility to define and use our own annotation types. An annotation type is declared like a normal interface declaration, with definition of methods corresponding to the properties of the annotations. Annotating an element consists in adding an at-sign (@) followed by the annotation type and a parenthesized list of attribute-value pairs in front of the declared element, before any other classical modifier (such as public, static, or final). An example of declarations and use of annotation is given in section3.2.

These typed metadata can be used by a lexical analyzer or processed at runtime by using the java reflection API. Sun Microsystems provide a command-line utility for annotation processing and a reflection API, named APT (Annotation Processing Tool), which can be used for automatic transformations by generating further source code. At runtime, the Java Virtual Machine gives the opportunity to introspect a model instance, and to retrieve information on the status of its components. At this level, the retrieved information can give the object type and the list of available methods for this object. This opportunity can be used to store metadata. Without annotations, it is possible to impose a specific argument signature to a method in order to obtain metadata at runtime. For example, a method such as *String getDescriptionOf(Field field)* can be used to obtain metadata about the *field* parameter. This is considerably limiting the compile time checking and in addition the major drawback of this method is that the developer needs to type a non-negligible amount of code to add constant information on an element. Annotations offer an easy way to add metadata, and give the advantage that this metadata is written just beside the element declaration. The reflection API gives methods to know if a declared element is annotated, and to fetch the content of annotations.

3.2 An implementation of the Observer design pattern

The Observer design pattern [1] defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The classic structure of this pattern is given by figure 1, we see two mains aspects of this pattern: the view (observer) aspect, and the document (subject) aspect. This patterns helps notifying observers of a state change in a document (subject), the number of observers being unknown.

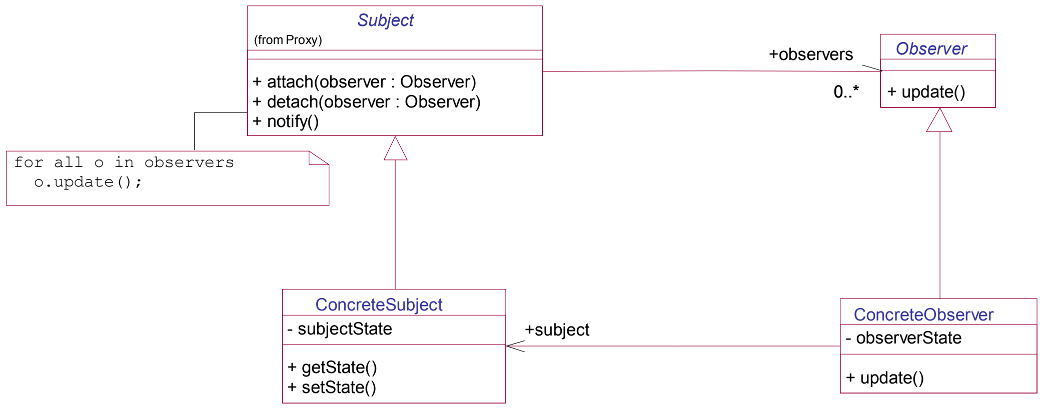


Figure 1. UML class diagram of the Observer pattern structure

Based on this structure we define a simple annotation that should be used in a class definition to declare a field as observable (which means a subject to observe). For example, in the code given below, the field *age* in the class *Individual* is annotated as observable, and this observable is described as “age in weeks”.

```
public class Individual {
     @Observable(description = "age in weeks")
        private short age;
     // ...
```

The principle is that when an observer is connected to a class, it can obtain the description and the value of an annotated field. Figure 2 illustrates the UML class diagram of this implementation. In order to speedup introspection operations, when an observable class is registered, the field references and the descriptions are stored once and for all by the *ObservableManager* in an object of type *Observable Handler* (see figure 2). The *ObservableManager* object is a singleton, meaning that only one instance could exist at runtime. This singleton is accessible by any other object that would add an observer on an observable object.

Every concrete observer will implement the *Observer* interface, mainly the *update()* method, which is called when

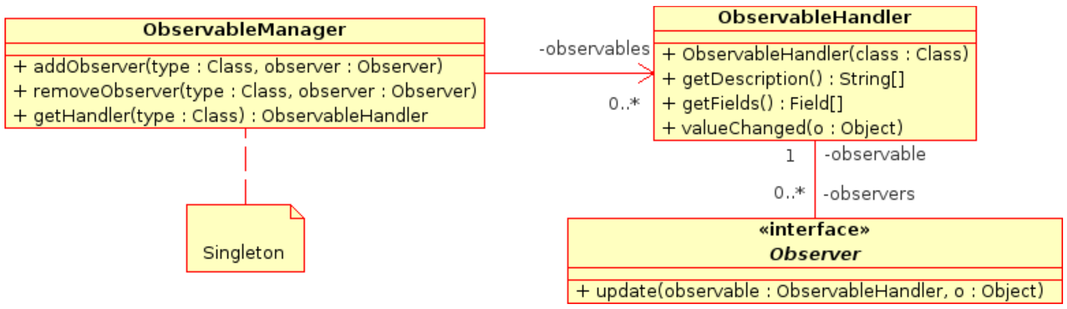

ObservableManager
+ addObserver(type : Class, observer : Observer)
+ removeObserver(type : Class, observer : Observer)
+ getHandler(type : Class) : ObservableHandler
-observables
0..*
ObservableHandler
+ ObservableHandler(class : Class)
+ getDescription() : String[]
+ getFields() : Field[]
+ valueChanged(o : Object)
1
-observable
0..*
-observers
Singleton
«interface»
Observer
+ update(observable : ObservableHandler, o : Object)

Figure 2. UML class diagram of our Observer implementation

a connected observable object has been updated. From the interface *Observer*, we have derivated two useful implementations. The first one, *SysoutObserver*, is the most simple observer, it prints the name and the value of each observable field, owned by the received object, on the standard output. The second one, *JFreeChartObserver*, is intended to use the JFreeChart software library (http://www.jfree.org/jfreechart) to plot the values of an observable on graphical charts. With the example of the class *Individual* precedently defined, an observer connected to an *Individual* object would be able to display sequential values of the *age* field, as text for the *SysoutObserver* or as a line chart for the *JFreeChartObserver*.

The updating event is fired by the method *valueChanged* of an *ObservableHandler*. This method can be called by the observed object itself when its variables are modified, or by an other object that is capable of modifying the observed object. The difference with the classic Observer pattern is that with our implementation, since an observer is connected to a class rather to an instance, several instances could be in fact connected to a same observer. In this case, variables coming from several instances should be sorted by the observer.

3.3 Using metadata to generate graphical user interfaces

The second example on which we focus aims to generate a graphical user interface (abbreviated as GUI) allowing the edition of object properties. Annotations give the opportunity to tune the generation process. We have called this tool OhOUI for Offhand Object User Interface. When the engine of OhOUI receives an object, the Java Reflection API is used to browse through the list of attributes. The engine has a list of basic converters able to build a user interface component from a set of predefined types. Java intrinsic types are supported (*int*, *short*, *byte*, *float*, *double*, *boolean* and *char*) and some other common types such as *java.io.File*, *java.awt.Color* and *java.lang.String* are also supported. If an attribute doesn't match a supported type, then this object is recursively browsed until the matching of a basic converter.

For example, if we consider the source code of figure 3 (from the *SysoutObserver* class), we have two attributes of type *boolean* and *java.io.File*. Each attribute type is natively supported and will be mapped with the corresponding converters to produce two graphical components. The resulting dialog box is shown in near to the code.

The *Description* annotation is used to generate a more significant name for GUI fields. Indeed, common attribute names are suffering from the constraints of the Java language syntax (space and special characters not allowed). In addition, a description will also be displayed as a tooltip when the mouse will be over the corresponding GUI object. As only the first field is annotated with a description, the GUI generator will use the attribute name for the label of the second field.

```
public class SysoutObserver implements Observer {
   @Description(  name     = "Use standard output",  tooltip  = "If desactivated, output will be writed to a file.")
   @Link(         action   = "disable",              target   = "filename")
         private boolean   sysout = true;            // First annotated attribute
   @Anchor(       id       = "filename")
         private File      outputFile = new File(""); // Second annotated attribute
}
```

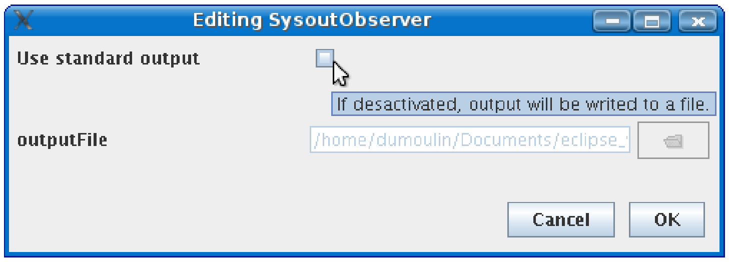

Editing SysoutObserver
Use standard output
If desactivated, output will be writed to a file.
outputFile
/home/dumoulin/Documents/eclipse_
Cancel
OK

Figure 3. Java class definition with annotations and corresponding GUI after the generation process.

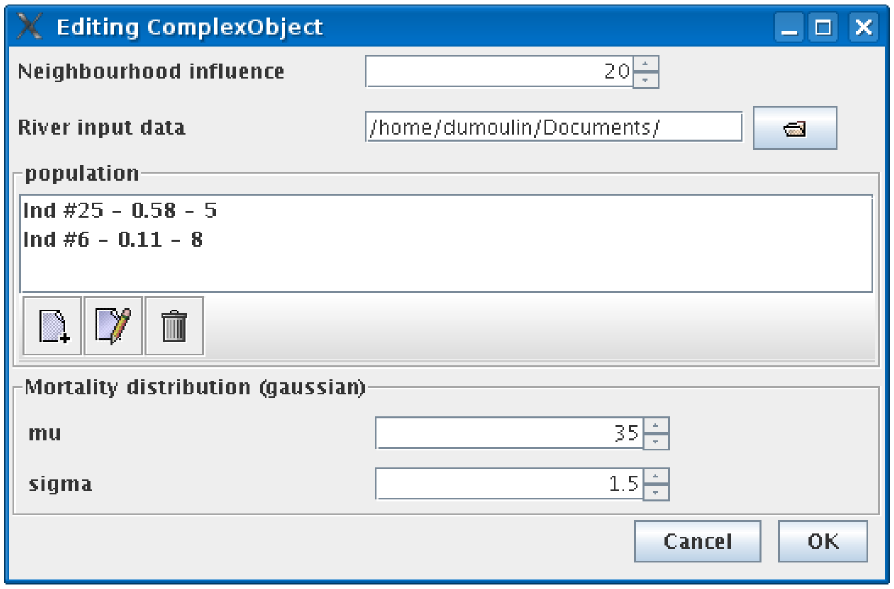


Figure 4. Example of a GUI generated from the definition of a complex type

Currently, we are working on additional converters to enable editing for complex data types, such as arrays and lists. A prototype demonstration is shown on figure 4.

4 BENEFITS FOR THE DEVELOPMENT OF AN MDA COMPLIANT IBM MODELING FRAMEWORK

4.1 Context

Model Engineering is showing a great potential, many companies use modeling elements to progressively and automatically generate executable codes. Thanks to the OMG (Object Management Group) who moved from its Object Management Architecture to a Model Driven Architecture [2], the software industry and many academic scientists are now giving more focus to model engineering. The Model Driven Architecture is a possible path, but various other similar approaches, such as CAMPaM (Computer Automated Multi-Paradigm Modeling) and ATOM3 fall in the model driven engineering philosophy for software development [3]. Microsoft proposed the concept of Software Factories to implement the model driven approach in Visual Studio (it relies on Domain Specific Languages, DSL) and the IBM vision of Model Driven Engineering is proposed in the now famous Eclipse Modeling Framework. All the previously cited approaches do have something in common which is based on meta-modeling.

The recent understanding of metamodeling relies on the fact that a metamodel is a specification or a definition for well defined models, though in the past many authors could state that a model of another model was a metamodel. A model can be roughly presented as a representation of a system, and transformation from a model to another model can be driven by a metamodel. Among many industrial examples of model transformation, the MDA QVT (Query View Transformation) is emerging. It has been successfully used in the field of automatic parsing (where a metamodel can be used to parse and reverse-engineer code).

We have recently developed a software framework named SimAquaLife. It is dedicated to the production of individual-based models (IBM) of freshwater fishes and we retained the model driven architecture. Reflective programming has also been used to produce models which rely on the metamodel proposed for the domain of IBM applied to freshwater fishes. This metamodel is represented by the UML class diagram on figure 5. Considering that the metamodel could be used for aquatic organisms (including fishes), our individuals are named *AquaNisms*. These aquanisms are grouped in an *AquaNismsGroup* that owns the dynamic processes. Each aquanism can be localized in the space by its position (class *Position*) and is annoted as observable in the *AquaNism* class definition. Thus, observers could be aware of the position of the individuals in order to print the traces of moves in a file, or to display them on an appropriate visual panel (see application in section 4.4).

4.2 Separation of concerns to hide implementation to the modeler

With the implementation proposed in the previous section, the modeler developing his model can distinguish between what is a configuration information specified by an annotation (and used by the reflection API to generate the GUI), from what is a model structure information. This separation of concerns should not be confused with aspect oriented programming (AOP) [4], because the AOP aims to be able to select at runtime which concerns will be effective.

For our purpose, we only want the reduction of the impact of some concerns that are not relevant during the model programming.

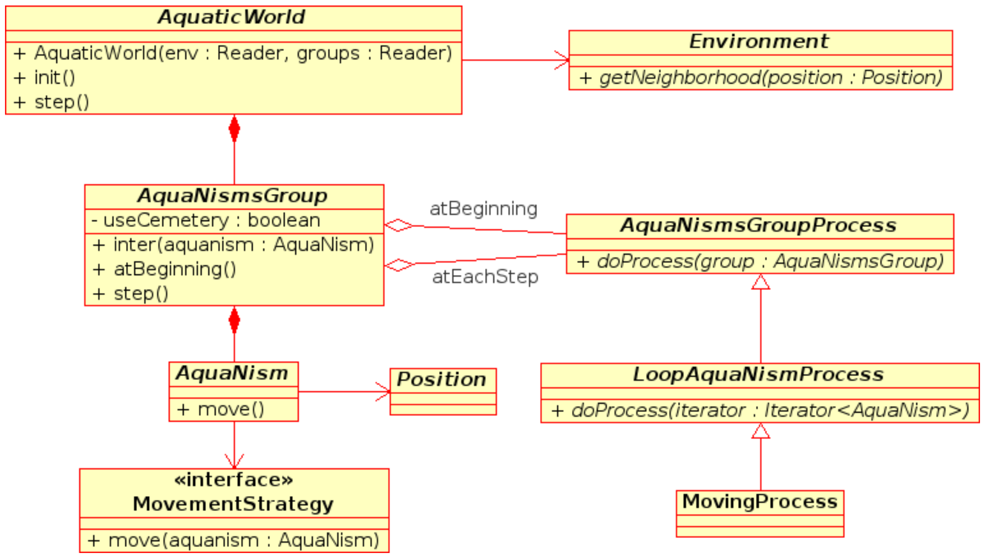


Figure 5. UML class diagram of the metamodel

4.3 Possibilities offered by metaprogramming

For computer scientists, metaprogramming is the ability to write programs able to generate or manipulate other programs (including themselves) [5]. This feature is sometimes used in very efficient C++ codes which benefit from template metaprogramming, the interested reader will find examples applied to game programming in the following book [6]. A list of public C++ metaprograms that could be useful for simulation can also be found on one of my former student home page (Pierre Chatelier http://ktd.club.fr/program mation/cpp-metaprog.php).

A metalanguage is the computer language in which we write a metaprogram, the programming language used for the generated code or for the code being manipulated is called the object-language. When the same programming language can be the object-language and the metalanguage, we have shown that we have the reflexivity property. In the previous section, we have seen that Java reflection and introspection were valuable language features to give the end-user a more contextual tool. Such features facilitated the generation of GUIs at runtime and it also saved a serious amount of programming time compared to what it would have taken to write all the code manually.

4.4 Results

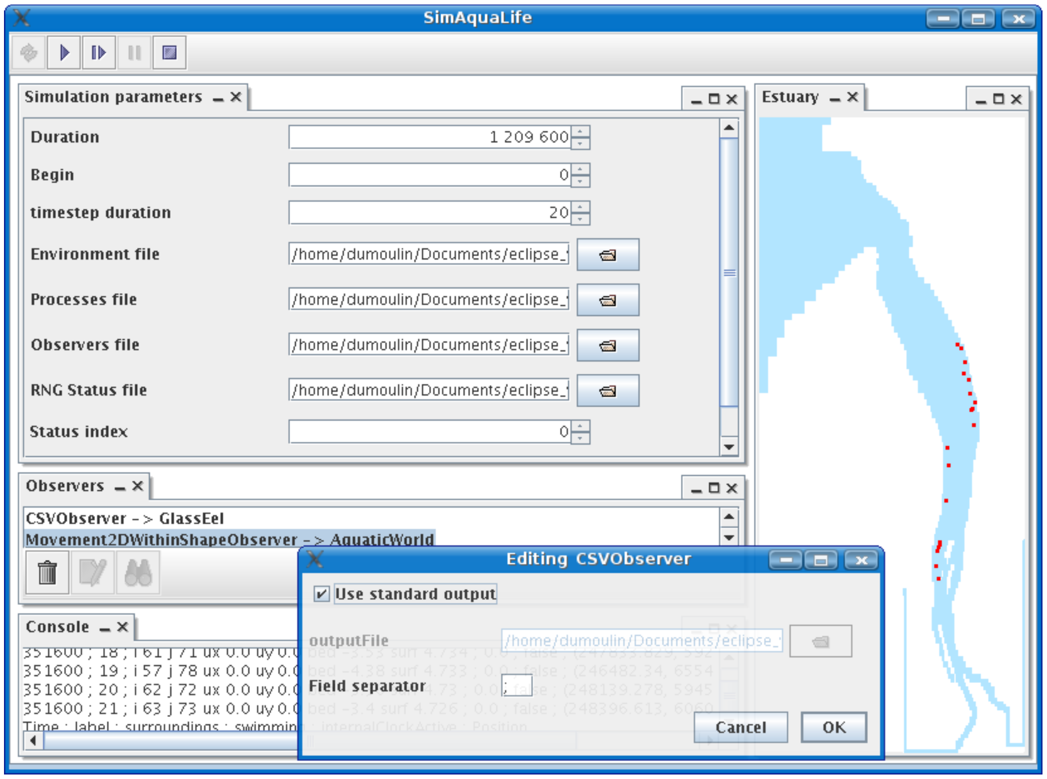
An individual-based model based on the SimAqualife framework was used to investigate the effects of physical and biological variables on the transport via tidal currents of glass eel (*Anguilla anguilla*) through the Gironde estuary [7]. This study aims to understand further the glass eel behaviour and to explain the river recruitment variability in this economically and ecologically important species. A screenshot of the final application at runtime is shown on figure 6. The main SimAquaLife window includes buttons for controlling the simulation, and several tabbed components. The top-left component provides the observers listening to observable objects. The observers listed could be edited, deleted or displayed if they have three buttons at the bottom of the "Observers" tab. The *CSVObserver*, present in the list of active observers, is an enhanced version of the *SysoutObserver* presented in the previous section, and prints observable fields separated by a character. At the bottom of the window, a console displayed all the messages that are sent by the simulation core component or by the plugged observers. On the right side, the tab titled "Estuary" is the display produced by the *Movement2DWithinShape Observer* observer. This display shows the fishes position in the estuary. The front dialog box, shows how we could edit the observer properties with the GUI generator.

This application allowed to assess the influences of spatial and temporal locations of departure and various synchronisation modes for vertical migration and transport paths.

5 CONCLUSION

In this paper we have considered the use of metadata to add information in a simulation model. This is particularly useful to modify the behavior of a model execution at runtime. We have achieved such model features using the Java Reflection API and metaprogramming techniques. We have given two practical examples illustrating the usefulness and simplicity of Java annotations. The first application demonstrates how such features can enhance the flexibility of the Observer design pattern. A second example presents how we can use annotations to generate automatically a kind of GUI Inspector corresponding to any Java class with the possibility to add at runtime new features and labels to the corresponding GUI panels. Such examples demonstrate the possibilities of the Java runtime introspection API for producing adaptable components.

The metaprogramming techniques exposed are used in the context of the development of an Individual Based Model in which the main GUI panels are automatically configured and generated at runtime, saving a considerable amount of manual coding (which is not often error prone). This work can be improved by supporting new types for automatic code generation, more complex converters could also be developed to handle lists, arrays, and major containers.



Duration	1 209 600
Begin	0
timestep duration	20
Environment file	/home/dumoulin/Documents/eclipse_
Processes file	/home/dumoulin/Documents/eclipse_
Observers file	/home/dumoulin/Documents/eclipse_
RNG Status file	/home/dumoulin/Documents/eclipse_
Status index	0

CSVObserver - > GlassEel
Movement2DWithinShapeObserver - > AquaticWorld

```
351600 ; 18 ; i 61 j 71 ux 0.0 uy 0.0 bed -3.53 surf 4.734 ; 0.0 ; false ; (247833.829, 592
351600 ; 19 ; i 57 j 78 ux 0.0 uy 0.0 bed -4.38 surf 4.733 ; 0.0 ; false ; (246482.34, 6554
351600 ; 20 ; i 62 j 72 ux 0.0 uy 0.0 bed -3.58 surf 4.73 ; 0.0 ; false ; (248139.278, 5945
351600 ; 21 ; i 63 j 73 ux 0.0 uy 0.0 bed -3.4 surf 4.726 ; 0.0 ; false ; (248396.613, 6060
Time ; label ; surroundings ; swimming ; internalClockActive ; Position
```

[✓] Use standard output
outputFile: /home/dumoulin/Documents/eclipse_
Field separator: ;

Figure 6. Screenshot of SimAquaLife running a model of glass eels migration in the Gironde estuary [6].

6 REFERENCES

[1] Gamma E., Helm R., Johnson R. and Vlissides J., "Design Patterns : Elements of Reusable Object-Oriented Software". Addison-Wesley, reading, MA, 1995.

[2] Kleppe A., Bast W. and Warmer J.B., "Mda Explained, the Model Driven Architecture: Practice and Promise", Addison-Wesley Professional, 2003, 192 p.

[3] Hans Vangheluwe H. and de Lara J., "Domain-Specific Modelling with AToM3.", In Juha-Pekka Tolvanen, Jonathan Sprinkle, and Matti Rossi, editors, The 4th OOPSLA Workshop on Domain-Specific Modeling, page 8 pp., October 2004. Vancouver, Canada.

[4] Kiczales G., Lamping J., Menhdhekar A., Maeda C.,. Lopes C, Loingtier J.-M. and Irwin J., "Aspect-oriented programming." In M. Aksit and S. Matsuoka, editors, Proceedings ECOOP, Springer-Verlag, volume 1241, pages 220–242. 1997.

[5] Cointe P., "Meta-Level Architectures and Reflection", Second International Conference," Reflection'99, Lecture Notes in Computer Science, Springer, 1999, 273 p.

[6] Deloura M., "Game Programming Gems II", Charles River Media, 2001, 575 p.

[7] Lambert P., Sottolichio A. and Dumoulin N., "Virtual experiments: the estuarine migration of glass eel with an individual-based model", 7th Indo-Pacific Fish Conference, Howard International House, Taipei, TWN, May 15-20, 2005, 1 p.

Nicolas Dumoulin got his MSc in Computer Science in 2002 from Blaise Pascal University and his French DEA from Rennes University in 2003. He obtained a fellowship to begin his PhD studies in the Laboratory of Engineering for Complex Systems at Cemagref. His research activities deal with the software implementation of an architecture to assist the development and analysis of individual-based models of freshwater-fishes.

David R.C. Hill received his Ph.D. degree in Object-Oriented Simulation in 1993 from Blaise Pascal University. D. Hill is currently Full Professor and co-manager of the ISIMA Modeling Institute. His current application domain concerns Life Science Simulation. (www.isima.fr/hill)