Using AOP In Discrete Event Simulation: A Case Study with JAPROSIM

Meriem Chibani
Department of Mathematics
and Computer Science
University of Oum El Bouaghi
Oum El Bouaghi, 4000, Algeria
Email: c.meriem@univ-oeb.dz

Brahim Belattar
Department of Computer Science
University of Batna
Batna, 5000, Algeria
Email: brahim.belattar@univ-batna.dz

Abdelhabib Bourouis
Department of Mathematics
and Computer Science
University of Oum El Bouaghi
Oum El Bouaghi, 4000, Algeria
Email: a.bourouis@univ-oeb.dz

***Abstract*—Japrosim is a discrete event simulation (DES) framework that has been developed for academic and industrial purposes based on object oriented paradigm. It contains several crosscutting concerns such as animation, steady state detection, keeping track of a simulation's state and graphical user interface (GUI). These concerns cross its modules and tend to decrease its modularity, understandability, maintainability, reusability, and testability properties. One of the latest offerings of software engineering domain is the aspect-oriented (AO) paradigm, which provides the ability to break free of object-oriented (OO) decomposition, and describe design with a greater degree of separation of concerns. In this paper, we identify Japrosim crosscutting concerns and propose practical AO solutions by means of the de facto AspectJ.**

***Keywords*—Crosscutting concerns, aspect oriented programming, discrete event simulation, Japrosim, AspectJ.**

I. Introduction

Simulation is "the process of describing a real system and using this model for experimentation, with the goal of understanding the system's behavior or to explore alternative strategies for its operation" [1]. The use of simulation improves understanding of systems behavior and the effects of interactions among their components.

Java PRocess Oriented SIMulation (JAPROSIM) is an open source and a free software simulation framework distributed under GNU Lesser General Public License (LGPLv3) since 2008. It is currently available under 1.4.1 version and is used for developing discrete event simulation models using a coroutine mechanism implementation. It is divided into eight main packages conceived and implemented for specific purposes: kernel, random, random.distributions, statistics, gui, utilities, animation and statistics.steady. It has been designed with object oriented approach and the Java language has been used for its implementation.

This library has been improved by adding several capabilities like graphical animation and steady state detection when others are under consideration like graphical modeling and explanation. On other hand, the evolution is an intrinsic property of software systems. As Japrosim is enhanced, modified and adapted to new requirements, the code becomes more and more complex and drifts away from its original design where different concerns could not be cleanly separated from its functional code. In the end, concerns are being intertwined in the library, which decreases its modularity and maintainability.

The aspect-oriented programming paradigm has emerged as the latest release of software engineering to offering the greatest degree of modularity, namely the separation of concerns through the modularization of crosscutting concerns. It could coexist with other approaches including the object oriented and several programming languages for AO systems has been developed as AspectJ (java extension), AspectXML (XML extension), AspectC++ (C++ extension), AspectC (C extension) [2]. Our work is focused on AspectJ, since it is the first practical AOP implementation based on Java, the most mature and the widely used in research area. An AOP system with an AspectJ implementation includes join points that are the places where the crosscutting actions take effect, while pointcuts select join points and collect context at those locations. The Aspect is the central unit in AspectJ, in the same way as a class is the central unit in Java. It could contain data, methods, and nested class members, like a normal Java class. The Advice is the code executed at a join point selected by a pointcut. An Advice can execute before, after, or around a join point. The body of the advice is much like a method body. It encapsulates the logic to be executed upon reaching a join point. Furthermore, the inter-type declaration (ITD) is a static crosscutting construct that alters the static structure of classes, interfaces, and aspects in the system. The weave-time declaration is another static crosscutting construct that adds weave-time warnings and errors when detecting certain usage patterns. Furthermore, one of the main elements of AOP is the "weaving" mechanism, which weaves together classes and aspects so that advice gets executed, inter-type declarations affect the static structure and weave-time declarations produce warnings and errors. AspectJ offers three weaving models: source weaving, binary weaving and load-time weaving [3]. The weaving may be static as implemented in AspectJ or dynamic e.g. AspectS implementation [4]. In this paper, we argue that some general and domain specific concerns of Japrosim are crosscutting over multiple modules. Therefore, this increases the complexity and reduces the maintainability. The use of the AOP, in this case, is justified by the fact that the domain requirements for simulation software are fairly

complex and the object oriented paradigm is not capable enough to deal with such complexity.

The remainder of the paper is organized as follows. Section 2 outlines the major related work. Section 3 identifies the Japrosim crosscutting concerns and gives the AOP solutions. Finally a conclusion is given in Section 4.

II. Related works

The use of the AO paradigm in DES depends on several considerations: (1) The level of the AO application (specification, design, or implementation phase); (2) The host methodology (multi-agent, components, or object oriented); (3) The aspect oriented language used (e.g. AspecJ, AspectS, or MAML); (4) The kind of the separated crosscutting concerns, that could be specific for simulation modeling domain, as steady state detection, or generic and found in any application as exception handling.

In the literature there are two DES frameworks that support AO paradigm. The first is Simkit, which is an open source tool based on the OO paradigm and event graph formalism for modeling. The authors in [5] proposed the AO solutions in terms of AspectJ for separating crosscutting concerns namely, the simulation termination rules, restoring a simulation run, and resource pooling. However, they lack to separate all crosscutting concerns, especially those which could be found in any mature framework as steady state detection and graphical user interface. The second is the component-based instrumentation framework OSIF [6] that uses the AOP paradigm to separate its DES models from the experimental frame to enable software reuse and evolution. The AOP has been used in the development of the Open Simulation Architecture (OSA) [7] for the same purpose as in OSIF. It allows better reuse of components in both sides, reuse of a given model with various scenarios or reuse of a given scenario with various models in order to save time, money and human effort. A simple use case of network security study has been used to illustrate the benefits of the previous technique.

For multi-agent simulation, the system discussed in [8] consists of two types of agents, a set for describing the simulation model and another for observational mechanisms. This collection of independent agents is interacting by discrete events where every agent has a schedule that generates its plan of activities. The system is executed on a framework that uses the OO paradigm to define its agent models and web technology to interact with the modelling and simulation environment. The kernel of this framework is the programming language MAML (Multi-Agent Modelling Language) which has the capacity of the dissociation between the model and the observational mechanisms, thanks to the aspect oriented paradigm, from the design phases until the implementation phase thereby increasing the maintainability of the system and decrease its complexity. MAML has the xmc compiler which generates Objective-C code from the MAML source code after weaving the model object and the observation object. Despite the richness of MAML syntax to support AOP, it remains in its infancy when compared to AspectJ.

In [9], the AO paradigm has been used to develop a multi-agent system dedicated to simulate physical phenomena. The MAFES system (Multi Agent Finite Environment System) consists of an environment in the form of a node matrix and a set of agents operating on these nodes. Aspects are used to assign tasks to agents by adding appropriate functionality to perform their task. In addition, these aspects weave the appropriate resources and attributes to the environments nodes. MAFES contains three other types of aspects for control, visualization and storing simulation results. The implementation of MAFES is based on AspectJ and makes the system generic to build versions for specific requirements (it is enough to weave appropriate aspects). The authors experiment their system with two phenomena. The first is heat exchange and a motion phenomenon. The second is heat exchange and crystallization.

AOP has been used for developing large simulation software systems. These simulation systems are based on the separation of concerns principle which increases their understandability and maintainability and keeping its performance and clarity. Among the proposed systems, we mention the disaster prevention simulation system and the conduit simulator.

In [10] a new aspect-oriented approach for disaster prevention simulation system (ABR) has been addressed. The proposed approach separates the core functionality of the simulation application from simulation crosscutting concerns thanks to horizontal decomposition (HD) method which relies on the AOP paradigm. The approach is implemented on AoSiF (Aspect-oriented Simulation Framework) which is an extension of distributed simulation framework (DiSiF) [11]. It uses the resource paradigm, actor based workflow modelling, web services and Grid computing as implementation technology and Java Annotations for declarative programming in addition to AspectJ for the aspect-oriented implementation. To demonstrate the applicability of the approach, two crosscutting concerns, namely distribution and tool integration are implemented. Unfortunately, concerns are not specific to the simulation modelling domain.

In [12] the authors implemented a conduit simulator system which uses the AOP paradigm at the code level. Considered crosscutting concerns as synchronization and order of execution, user interface (UI) and logging, are written in different aspect-specific languages (ASLs). The simulator has a modular aspect-weaver mechanism that offers the generality of a general-purpose aspect language without losing the ability and advantages of defining aspects in aspect-specific languages. Furthermore, the conduit simulator crosscutting concerns are not specific to the simulation modelling domain.

The exploitation of the AOP is not limited for discrete event simulation. It affects many informatics domains as security and software engineering approaches, we could mention:

In [13], an Automated data Collection approach for Usability Evaluation in Early Stages of Application Development is discussed. This approach is based on AOP and uses aspects for collecting usability data in an adaptive and self-contained fashion without having to instrument the target application or its platform.

In [14], authors investigate the general aspect-oriented software security development process, and propose an Eclipse-based software security aspect development tool. This tool provides reusability of software security aspects, and improves security developer's productivity.

III. Object Oriented Japrosim Design

A large research effort has been devoted to enrich mainstream languages as C, C++, Java, Python with simulation capabilities. The most common choice is to provide the additional simulation functionality through a software library. Independently of the architectural level at which they are provided (application, library, language), the simulation capabilities embody a world view for their users. The world view is essentially the set of concepts that constitute the basic elements available to the modeler to compose and to specify the simulation. The diverse world views are functionally equivalent, but differ in expressive power and in terms of computational efficiency. The idea of building process-oriented simulations using a general purpose object-oriented programming language is not original and several tools were developed in this way. For example, both of CSIM++ and YANSL are based on C++, while PsimJ , JSIM are based on Java. Discrete Event Simulation tools written in Java, like PsimJ and SSJ are well designed and freeware libraries but not open source [15].

JAPROSIM is an object-oriented simulation library, free and open source that adopts the popular process- interaction worldview. The library is implemented in Java programming language allowing deep access to its powerful features. The library is divided into packages to organize the collection of classes into important functional areas. The library is divided into eight main packages:

- kernel: a set of classes dealing with active entities, scheduler, queues and resources.
- random: contains classes for uniform random stream generation.
- random.distributions: contains a rich set of classes for useful probability distributions.
- statistics: contains classes representing intelligent statistical variables.
- gui: a set of graphical user interface classes to use for project parameterization, trace and simulation results presentation.
- Utilities: a set of useful classes for express model development.
- statistics.steady: useful to detect the steady state.
- animation: a set of classes used to provide a real time animation of simulation models.

The kernel package is at the heart of JAPROSIM. It is made up of classes dealing with active entities, scheduler, queues and resources as shown in figure 1. The coroutine like mechanism is implemented through SimProcess, Scheduler, StaticEntity and Entity classes. A coroutine program is a collection of coroutines which run in quasi-parallel with one another. Each coroutine is an object with its own execution state, so that it may be suspended and resumed. JAPROSIM was putting a great emphasis into following the semantic of SIMULA but the design itself is not close to it. The advantage of this approach is that design is simpler without explicit coroutine class support and the semantics of facilities that are well-known and thoroughly tested through many years use of SIMULA are completely supported. Native support for multithreaded execution is a fundamental aspect to the implementation of a natural process-oriented modeling capability in Java. Every active entity's life cycle is executed in a single separate thread [16].

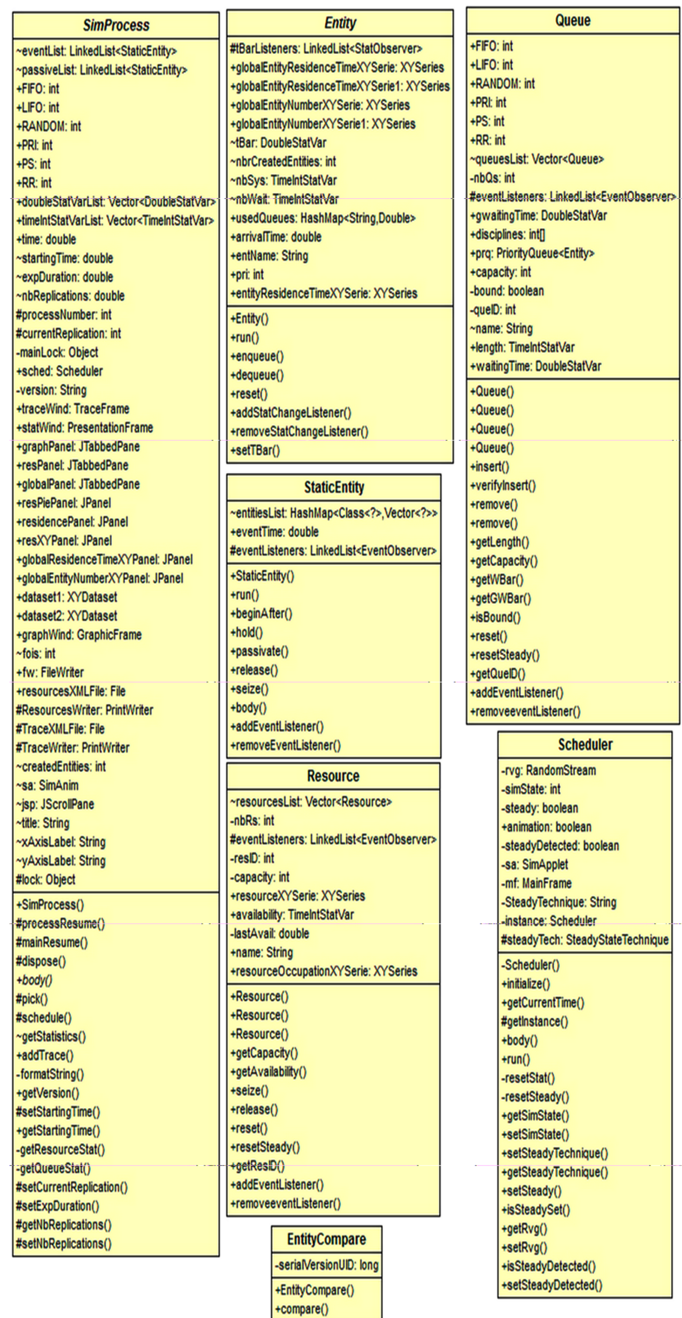

Fig. 1. The Kernel class diagram.

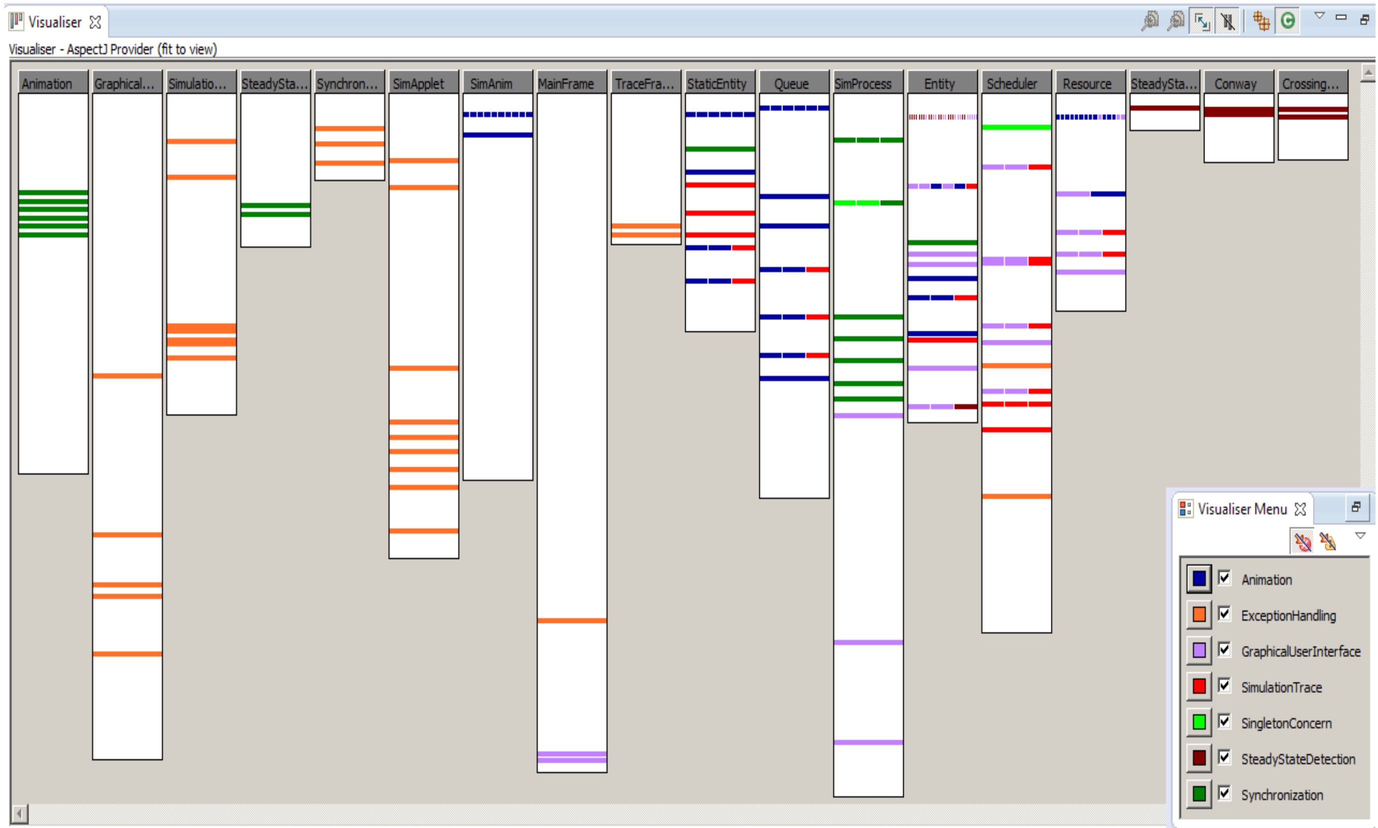

Fig. 2. The AO Japrosim aspects affect in crosscutting concerns separation.

IV. The AO Japrosim version

The AO Japrosim is enhanced with the *crosscutting.concerns* package which consists of seven aspects implemented to deal specific crosscutting concerns: SingletonConcern, Animation, ExceptionHandling, GraphicalUserInterface, SimulationTrace, SteadyStateDetection and Synchronization. Every aspect gives a solution to separate one of the main Japrosim crosscutting concerns which pollute the framework, as shown in figure 2.

A. Singleton Concern

In classical OO, the Scheduler class uses the “Singleton” pattern to prevent multiple instances and ensure that the event list management is done exclusively by a unique thread. Japrosim ensures that by declaring its constructor as private, providing a public method namely, getInstance() to return the single existing instance, and saving the single existing instance as static member variable. The SingletonConcern aspect is the proposed solution which includes an around advice that applies at the moment of the constructor call that has the similar role of getInstance() method without the need to declare the singleton constructor as private. Furthermore, the Scheduler instance is saved in a static member introduction variable declared inside the aspect.

B. Graphical Animation

The SimAnim class is part of the animation package used to provide a real time animation. It recuperates useful data by means of the “EventObserver” interface. Moreover, each of the “Queue”, “Resource” and “StaticEntity” classes register listeners of “EventObserver ” type to inform them in case of event occurrence. This mechanism is ensured by the observer pattern. The Animation aspect is proposed as a solution as shown in figure 3 where a snipped code is given. It separates these elements by providing the link between subject and observer using inter-type declaration (ITD) in the form of “EventObserver” interface declaration inside the aspect, member introduction, and type-hierarchy modification which are infected subjects and SimAnim class respectively.

C. Steady State detection

The output data collected during the warming-up period of a simulation can be misleading and bias the estimated response measure. Thus, the removal of initialization bias is important for obtaining accurate performance estimators. There are five categories of methods for steady state detection, graphical, heuristic, statistical, initialization bias tests, and hybrid methods [17]. Japrosim classic version offers two methods that are Conway and Crossing the Means by means of factory and observer design patterns for steady state detection. The observer elements are tangled in the Entity,

```
public static void Resource.addEventListener(EventObserver listener) {
    Resource.eventListeners.add(listener);
}
public static boolean Resource.removeEventListener(EventObserver listener) {
    return Resource.eventListeners.remove(listener);
}
public static void Queue.addEventListener(EventObserver listener) {
    Queue.eventListeners.add(listener);
}
public static boolean Queue.removeEventListener(EventObserver listener) {
    return Queue.eventListeners.remove(listener);
}
pointcut ResourcePointcut(Resource r, String s, int cap) :
    execution (public Resource.new(String , int ))
    && (this(r) && args(s,cap));
pointcut SEntityP1(StaticEntity se, Resource res, int units) :
    (execution(public void StaticEntity.release(Resource,
     int)) && this(se)) && args(res, units);
pointcut SEntityP2(StaticEntity se, Resource res, int units) :
    (execution(public void StaticEntity.seize(Resource,
    int)) && this(se)) && args(res, units);
pointcut spointcut(StaticEntity se) :
    (call(protected void dispose()) && target(se))
    && withincode(public void run());
pointcut Ep1(Queue q) : execution(public void Entity.enqueue(Queue))
                        && args(q);
pointcut Ep2(Queue q) : execution(public void Entity.dequeue(Queue))
                        && args(q);
pointcut Qp1(Entity e) : execution(public * Queue.*nsert(Entity))
                        && args(e);
pointcut Qp2(Entity e) : execution(public void remove(Entity))
                        && args(e);
after(EventObserver sim) : (execution(public SimAnim.new())
                        && this(sim)) {
    StaticEntity.addEventListener(sim);
    Resource.addEventListener(sim);
    Queue.addEventListener(sim);
}
```

Fig. 3. Animation aspect snipped code.

SteadyStateTechnique, Conway and CrossingTheMean classes as shown in figure 4 where highlighted elements contain the necessary code to implement this instance of the observer pattern. The SteadyStateDetection aspect is proposed to solve the problem.

D. Synchronization of Simulation Processes

Japrosim implements the coroutine mechanism through SimProcess, Scheduler, StaticEntity and Entity classes. A collection of threads are run in quasi-parallel under the Scheduler thread supervision. Each coroutine is an object with its own execution state, so that it may be suspended and resumed. At any instance of real time only one coroutine is active. The method processResume(Entity e) is called by the scheduler to reactivate a simulation process and mainResume() is called by a simulation process to reactivate the scheduler. Each simulation process has its own lock object. The scheduler has the mainLock object. Locks are used in combination with wait() and notify() to synchronize the implementation threads. A thread which calls any of the previous methods will block on its own lock after notifying the appropriate one. At the end of its life cycle, a simulation process calls automatically the dispose() method to reactivate the scheduler without blocking itself. So the corresponding thread could be

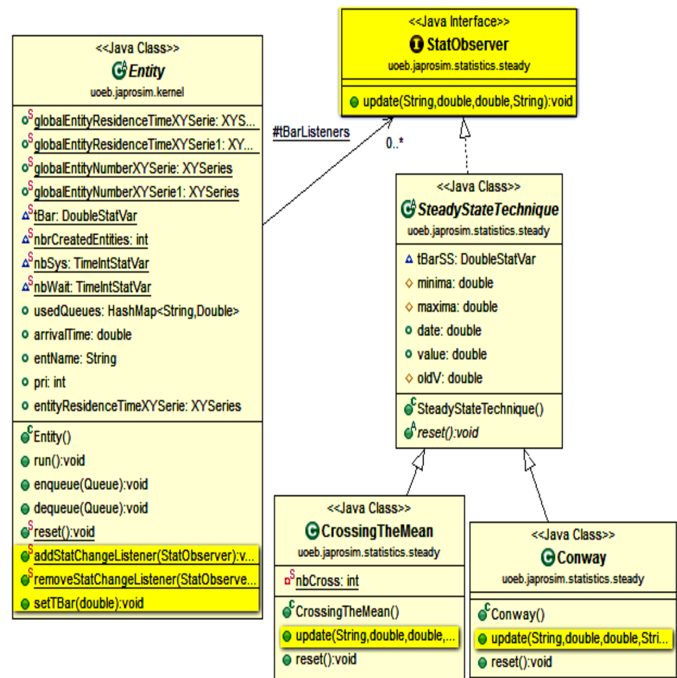

Fig. 4. The Steady State detection based on the Observer pattern.

terminated [16]. Elements that ensure the coroutine mechanism are the processResume(Entity e), mainResume() and dispose() methods in addition to the mainLock and lock objects. These elements are respectively separated in a single Synchronization aspect that clears the design and increases understandability.

E. Keeping Simulation Track

To save simulation trace, Japrosim generates three files. One is in text format (trace.txt) and the two others are conforming to XML format (trace.xml and Resource.xml). Each event executed during the simulation implies updating these files which makes this concern crossing the Japrosim functional modules. As an example, the public void remove(Entity) method removes the specified entity from the queue then calls the addTrace(String) method and the TraceWriter to save this event in both trace.txt and trace.xml files as illustrated in figure 5. The SimulationTrace aspect provides the solution illustrated by the after advice as shown in figure 6.

F. Graphical user interface (GUI)

The Japrosim gui package includes a set of classes (TraceFrame, PresentationFrame, MainFrame, and GraphicFrame) to use for project parameterization, for instance number of replications, experiment duration, trace and simulation results presentation. Despite the benefits and flexibility provided by the GUI concern, its crosscutting nature leads to code pollution. It decreases the cohesion, specifically inside Japrosim kernel classes. As an AOP solution, the GraphicalUserInterface aspect is proposed.

```
public void remove(Entity e) {
    if (this.prq.remove(e)) {
        this.waitingTime.update(SimProcess.time
                - e.usedQueues.get(this.toString()).doubleValue());
        Queue.gwaitingTime.update(SimProcess.time
                - e.usedQueues.get(this.toString()).doubleValue());
        Entity.nbWait.decrement();
        length.decrement();
        SimProcess.addTrace("      "+e.getName()+" leaves queue "
                + this.name + "\n");
        SimProcess.TraceWriter.println("<Event "+" Date = \""
                +SimProcess.time+"\" Subject = \""+e.getName()
                +"\" Identifier = \""+e.getId()+"\" Action = \""
                +"leaves"+"\" Object = \""+this.name + "\"/>");
        if (Scheduler.animation) {
            int p = Integer.parseInt(e.getName().substring(
                    e.getName().lastIndexOf(" ") + 1));
            ListIterator<EventObserver>it=eventListeners.listIterator();
            while (it.hasNext()) {
                it.next().update(e.getClass().getSimpleName(), p,
                        this.getQueID(), 0, 2, SimProcess.time);
            }
        }
    }
}
```

Fig. 5. The remove() method with the simulation track crosscutting concern highlighted.

```
after(Queue q, Entity e) : (this(q)
    && get (public TimeIntStatVar length))
    && (cflowbelow(p22(e))
    && withincode(public void remove(Entity))) {
    addTrace("    " + e.getName() + " leaves queue " + q.name + "\n");
    TraceWriter.println("<Event " + " Date = \"" + SimProcess.time
        + "\" Subject = \"" + e.getName() + "\" Identifier = \""
        + e.getId() + "\" Action = \"" + "leaves" + "\" Object = \""
        + q.name + "\"/>");
}
```

Fig. 6. The remove() advice inside the SimulationTrace aspect.

G. Simple Synchronization

Mutual exclusion synchronization restricts concurrent activities in critical sections to protect them against data inconsistency due to simultaneous access for writing. In Java, the synchronization is implemented by using the synchronized modifier at the method level or the synchronized(Object) construct at the instruction or block level. In an object oriented multithread programming, an object could be accessed by many threads simultaneously. In consequence, data conflicts could occur if applications are not prepared to deal with concurrency [18]. Thus, OO Japrosim methods which have a critical property are enhanced with the synchronized keyword as the getInstance() method in Scheduler class. This tends to pollute the Japrosim functional code. In order to overcome this problem, an around advice inside the synchronization aspect is developed to ensure methods synchronization, as shown in figure 7 by using a shared aspect lock.

H. Exception Handling

The initial solution is proposed in [19]. Currently, after several significant improvements, the ExceptionHandling aspect contains five pointcuts and five advices that deal all catch blocks in Japrosim library as illustrated by the snipped code in Figure 8.

```
Object around() :
    (execution(protected static void SimProcess.schedule(StaticEntity))
        || execution(protected static StaticEntity SimProcess.pick()))
        || (((execution(* *.add*Listener(..))
        || execution(* *.remove*Listener(..)))
        || (call (public Scheduler.new())
        && !within(SingletonConcern)))) {
    synchronized (this) {
        return proceed();
    }
}
```

Fig. 7. The synchronization aspect snipped code for methods synchronization.

```
pointcut pointcut5(Exception e) : (handler(Exception+) && args(e))
                                  && (!(pointcut4(Exception))
                                  && !pointcut3(UnsupportedEncodingException))
                                  && (!(pointcut2(Exception))
                                  && !(pointcut1(FileNotFoundException)));
before(FileNotFoundException fnf) : pointcut1(fnf) {
    System.err.println("Resources XML File couldn't be created");
    System.err.println(fnf);
    System.exit(1);
}
before(Exception ex) : pointcut2(ex) {
    System.err.println("Cannot install " + MainFrame.PREFERRED_LOOK_AND_FEEL
            + " on this platform:" + ex.getMessage());
}
before(UnsupportedEncodingException uee) : pointcut3(uee) {
    System.err.println("utf-8 not recognized nor suported !");
    System.err.println(uee);
    System.exit(1);
}
before(Exception ie) : pointcut4(ie) {
System.err.println("Error closing file.");
ie.printStackTrace();
}
before(Exception e) : pointcut5(e) {
    e.printStackTrace();
}
```

Fig. 8. The ExceptionHandling aspect snipped code.

V. CONCLUSION

In this paper, we have proposed the solutions for separating Japrosim crosscutting concerns using the aspect oriented programming paradigm, specifically AspectJ language. The AO version grew out from the exploitation of the aspect oriented programming paradigm in the simulation modeling field which is considered as a promising research area that provides many benefits in term of design quality improvement. In future work we plan to:

- Compare the classic Japrosim version with the proposed one in order to count the main improvements after the aspect oriented paradigm implementation.
- Spread the use of the AOP at the earlier development phase as modeling level using aspect oriented software development (AOSD) techniques.

REFERENCES

[1] R. E. Shannon, Systems simulation the art and science, Prentice Hall, Englewood Cliffs. 1975.

[2] A. Kumar, “Analyse and design of metrics for aspect oriented systems”. Doctorate Thesis. School of Mathematics and Computer Application, Thapar University, Patiala-147 004 (Punjab), India. 2010.

[3] R. Laddad, “AspectJ in action: enterprise AOP with Spring applications”. Second Edition, Manning Publications, Greenwich, USA (2009).

[4] R. Hirschfeld, “Aspect-Oriented Programming with AspectS”, in: Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe 2002, Erfurt, Germany.

[5] A. U. Aksu, F. Belet and B. Özdemir, “Developing aspects for a discrete event simulation system”. In: Proc. Third Turkish Aspect-Oriented Software Development Workshop, pp. 84–93, Bilkent University, Ankara, Turkey (2008).

[6] J. Ribault, O. Dalle, D. Conan and S. Leriche, “OSIF: A framework to instrument, validate, and analyze simulations”. In: Proc. L. Felipe Perrone, Giovanni Stea, Jason Liu, Adelinde Uhrmacher, Manuel Villn-Altamirano. (eds.) 3rd International Conference on Simulation Tools and Techniques, SIMUTools ’10, pp. 56–56, Malaga, Spain (2010).

[7] J. Ribault and O. Dalle, “Enabling advanced simulation scenarios with new software engineering techniques”. In: 20th European Modeling and Simulation Symposium (EMSS 2008), Briatico, Italy (2008).

[8] L. Gulyás and T. Kozsik, “The use of aspect-oriented programming in scientific simulations”. In: Jaan Penjam, editor, Software Technology, Fenno-Ugric Symposium (FUSST’99), pp. 17–28, Tallinn, Estonia (1999).

[9] S. Bieniasz, S. Ciszewski and B. Śnieżyśki, “Multi-agent simulation of physical phenomena by means of aspect programming”. In: Proc. The 6th international conference on Computational Science. Vassil N. Alexandrov (eds.) ICCS 2006. LNCS, vol. 3993, pp. 759–766, Springer-Verlag, Heidelberg (2006).

[10] T. B. Ionescu, A. Piater, W. Scheuermann and E. Laurien, “An aspect-oriented approach for the development of complex simulation software”. Journal of Object Technology, Vol. 9, no. 1 (January 2010), pp. 161–181, doi:10.5381/jot.2010.9.1.a4. (2010).

[11] A. Piater, T. B. Ionescu and W. Scheuermann, “A distributed simulation framework for mission critical systems in nuclear engineering and radiological protection”. International Journal of Computers Communications & Control, ISSN 1841–9836, vol. 3, no. 1, pp. 448–453. (2008).

[12] J. Brichau, K. Mens and K. D. Volder, “Building composable aspect-specific languages with logic metaprogramming”. In: Proc. The 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering GPCE’02, Vol. 2487, pp. 110—127, Springer—Verlag, London (2002).

[13] Y. Tao. “Automated Data Collection for Usability Evaluation in Early Stages of Application Development”. Proceedings of the 7th WSEAS Int. Conf. on APPLIED COMPUTER & APPLIED COMPUTATIONAL SCIENCE (ACACOS ’08), pp. 532-540, Hangzhou, China, 2008.

[14] C. Wang, M. Huang. “AOSecBuilder: An Aspect-Oriented Security Component Development Toolkit”. Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications,pp. 470-473, Elounda, Greece, 2006.

[15] B. Belattar and A. Bourouis, “Ascertaining Important Features of the JAPROSIM Simulation Library”, Proceedings of the 2013 EUROPMENT International Conference on Systems, Control, Signal Processing and Informatics, Rhodes Island, Greece, 2013.

[16] A. Bourouis and B. Belattar, “JAPROSIM: A Java Framework for Discrete Event Simulation”. Journal of Object Technology, vol. 7, no. 1, pp. 103–119.(2008).

[17] K. Hoad, S. Robinson and R.Davies, “Automating Warm-Up length estimation”, Proceedings of the 2008 Winter Simulation Conference , pp. 532-540, 2008.

[18] C. A. S. D. Cunha, “Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms”, MSc. Thesis. University of Minho, Braga. 2006.

[19] M. Chibani, B. Belattar and A. Bourouis, “Toward an aspect-oriented simulation”. International Journal of New Computer Architectures and their Applications (IJNCAA), Vol. 3(1), pp.1-10, 2013.