

# Towards the Use of Sequence Diagrams as a Learning Aid

João Paulo Barros<sup>\*,+</sup>, Luís Biscaia<sup>\*</sup>, and Miguel Vitória<sup>\*</sup>

<sup>\*</sup>*LabSI<sup>2</sup> & ESTIG, Instituto Politécnico de Beja, Rua Pedro Soares, Beja, Portugal*  
<sup>+</sup>*UNINOVA-CTS, Portugal*

joao.barros@ipbeja.pt, lcds.biscaia@gmail.com, migueljvt@hotmail.com

## 1 Introduction

Compared to imperative programming, object-oriented programming brings additional complexities. These complexities are especially challenging for the novice and, as a consequence to the teacher. Hence, it is no surprise that the teaching and learning of object-oriented programming is an extremely popular topic in computer science education research.

This work in progress paper presents the objectives and structure of a tool under development for novice object-oriented programmers that intends to ease code understanding. That is accomplished through the use of sequence diagrams, one of the most popular behavior diagrams in the Unified Modeling Language (OMG, 2011), the *de facto* standard for object-oriented modelling. More specifically, the tool allows the generation of execution traces as sequence diagrams: for a given program run, the student is able to visualize the respective execution as a sequence diagram. Next, we present the Java2Sequence tool.

## 2 The Java2Sequence tool

Object-oriented programming brings additional layers of indirection right from the beginning. More specifically, operations can be executed by classes or by objects and variables can have values or addresses of values. Sequence diagrams offer a simple and visual representation for these indirections, namely the class-object duality, as well as for object creation and method calling. More specifically, the Java2Sequence tool has the following objectives, which when taken together are not, to the best of our knowledge, fulfilled by any available tool:

1. To ease program comprehension for novices learning object-oriented programming using Java;
2. The dynamic generation of UML sequence diagrams, a well-known type of behavior diagram, but enriched with informative extra annotations, while offering a visual representation for several fundamental concepts, namely classes and objects lifelines, object creation, instance and class method calling, parameters, return values, and recursion;
3. A step by step execution of Java code;
4. The generated sequence diagrams should allow (a) filtered views for the simplification and reduction of the resulting diagrams and (b) the visualization of each thread role in program execution.
5. Integration with other tools for teaching programming at the introductory level; we foresee the BlueJ programming environment (Kölling, 2011) and the Violet UML editor (Horstmann and de Pellegrin, 2011); another possibility will be a UMLGraph specification as it supports sequence diagrams (Spinellis, 2008);

Presently, objectives 3, 4b, and 5 are still not completed.

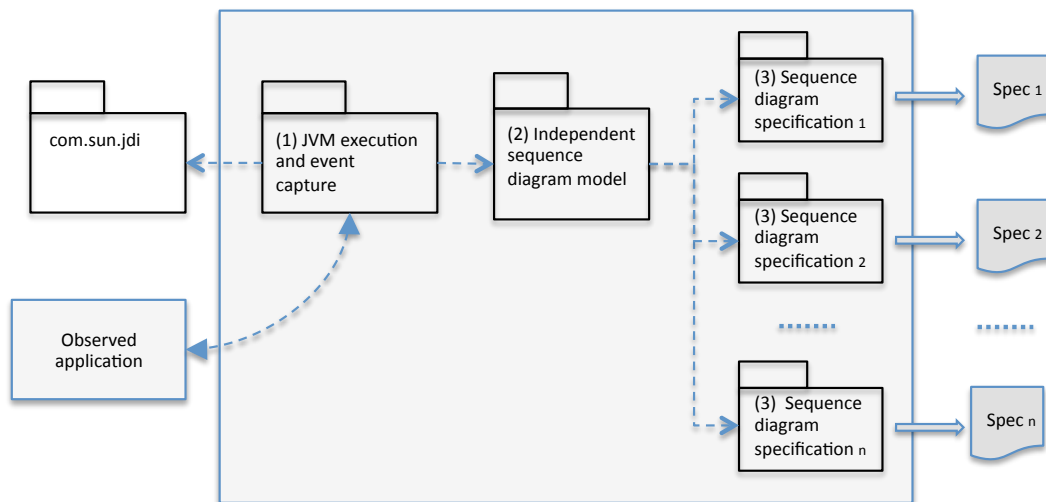
### 2.1 Annotations

The UML specification is quite short regarding annotations for sequence diagrams. Hence, an important addition is the definition of meaningful and informative annotations for each

diagram element namely lifelines, activations, and arrows. When these annotations are too long to be presented, they are hidden and replaced by a small plus sign. They are made visible putting the mouse over the plus sign. The examples in the Section 2.3 exemplify some of these annotations. Next, we present a brief overview of the tool architecture.

## 2.2 Architecture

The tool is structured in three parts: (1) the core that executes the program being analyzed and handles events from the observed application; (2) the internal representations for the sequence diagram; (3) one or more packages that generate different model specifications. These latter specifications can be textual formats (e.g. XML files), allowing interchange and visualization in other tools, or graphical representations requiring distinct implementations (e.g. Java Swing based diagrams). Fig. 1 illustrates the three part decomposition, the relation to the observed application and the resulting specifications.



**Figure 1:** Tool structure showing relation with the observed application and generated specifications.

## 2.3 Examples

Here, we present two short examples. The first was more briefly presented in (Barros et al., 2011). It uses a simulator for a vending (dispenser) machine: the `main` method creates a `Dispenser` object, which creates a `MoneyBox` object. Then, the `main` method asks the `Dispenser` object to add a new `Product` object. Finally, the `main` method asks the `Dispenser` object to execute the `insertCoin` operation with an `int` parameter with value 20. This simple example also illustrates object delegation as the `insertCoin` operation is delegated to the aggregated `MoneyBox` object. It is important to stress that this kind of programs where several objects interact, are especially interesting for sequence diagram, as it becomes possible to visually show the different objects and classes, as well as the respective interactions. Nevertheless, after this example, we present a shorter one, with no objects. The intent is to show that recursion can also be visualized as multiple subactivations of a given instance method.

Listing 1: Dispenser machine

```
package ipbeja;

public class Dispenser {
    private List<Product> products; // products in machine
```

```

private MoneyBox moneyBox; // handles money

public Dispenser() {
    this.products = new ArrayList<Product>();
    this.moneyBox = new MoneyBox();
}

public void addProduct(Product p) {
    this.products.add(p);
}

public int insertCoin(int coin) {
    return this.moneyBox.insertCoin(coin);
}

public static void main(String[] args) {
    Dispenser dispenser = new Dispenser();
    dispenser.addProduct(new Product("cookies", 100));
    dispenser.insertCoin(20);
}
// ...
}

```

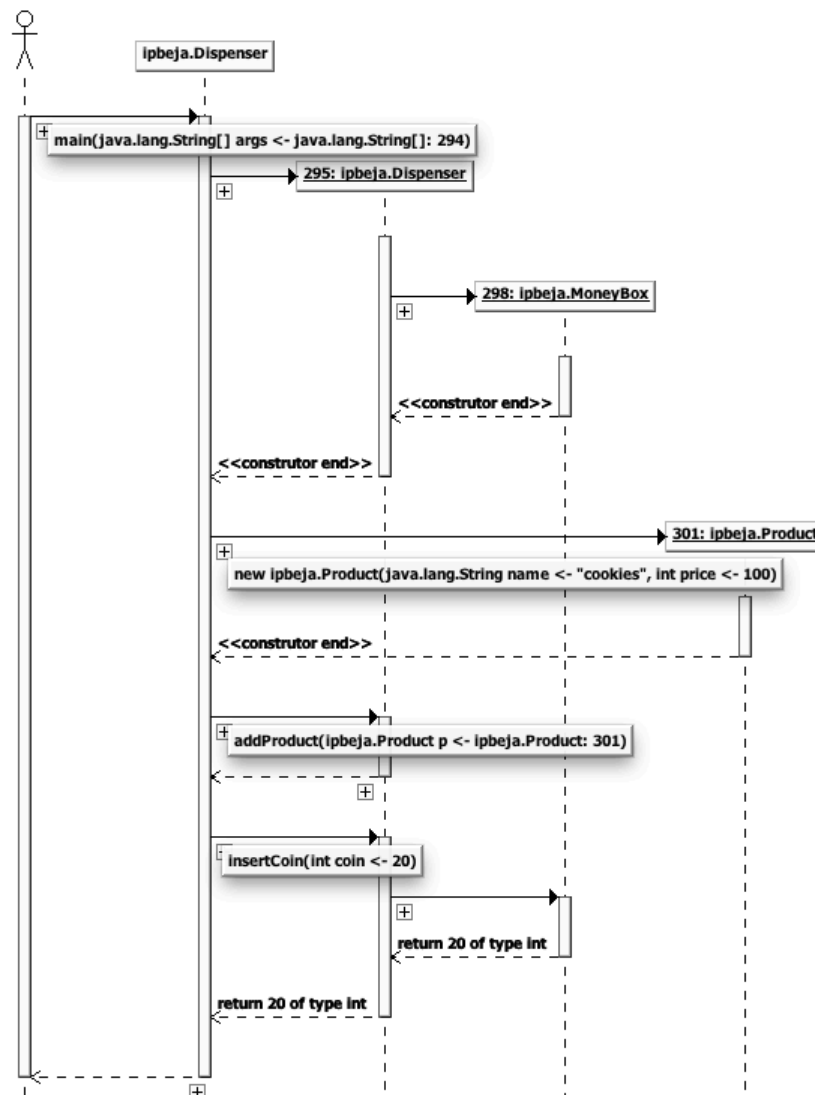


Figure 2: Sequence diagram generated from Listing 1.

The second example in Listing 2 and Fig. 3 shows recursive calls for a short Fibonacci computation.

Listing 2: Fibonacci calculation

```

package fibonacci;

public class Fibonacci {

    public static void main(String[] args) {
        Fibonacci s = new Fibonacci();
        s.fib(2);
    }

    public int fib(int n) {
        if(n == 0 || n == 1) {
            return n;
        }
        else {
            return fib(n - 1) + fib(n - 2);
        }
    }
}

```

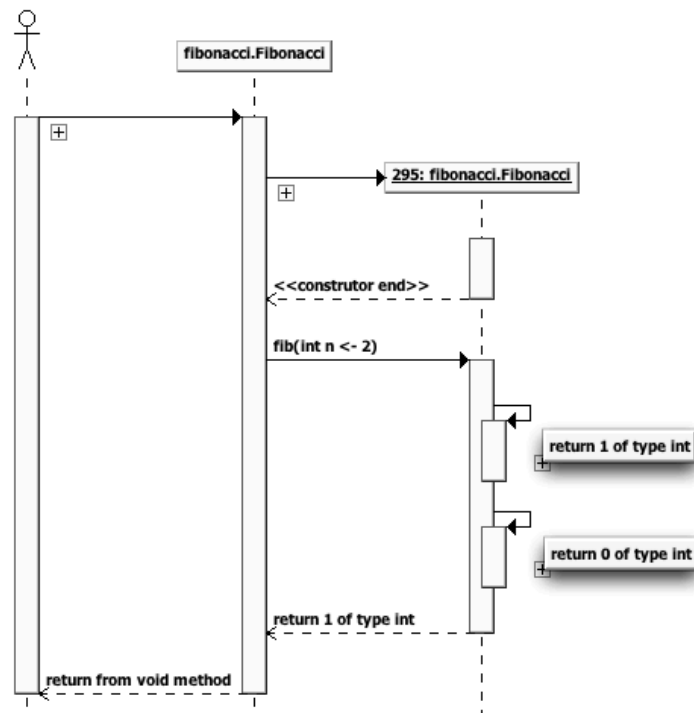


Figure 3: Sequence diagram generated from Listing 2.

### 3 Related Work

The use of UML in an introductory learning context has been the subject of numerous contributions. Especially due to the size and complexity of the UML specification and even of each of its diagrams, some significant efforts have been made towards the development of simpler and more user friendly UML tools, e.g. (Turner et al., 2005; Ramollari and Dranidis, 2007; Horstmann and de Pellegrin, 2011). Here, we reference related articles and freeware tools more directly related to the generation of UML sequence diagrams from Java source code.

The JACOT tool (Leroux et al., 2003) emphasizes the visualization of thread states using several different diagrams. Some other proposals also emphasize the visual understanding of

concurrency, e.g. (Malnati et al., 2008) and (Mehner, 2002). The latter is presented as work in progress and uses the UML CASE Tool Together (from [www.togethersoft.com](http://www.togethersoft.com), now part of Borland.com) for deadlock detection and analysis based on tracing. Differently, regarding concurrency, Java2Sequence emphasizes method calling. Then, for each method call, it should be possible to see the responsible thread.

Oechsle and Schmitt (2002) present the JAVAVIS tool, which uses the Java JDI interface and allows step by step drawing of UML sequence and object diagrams. It also allows the visualization of each thread role. Yet, it does not foresee integration with other educational tools and, to the best of our knowledge, it is not available.

The JAN tool relies on code tagging to generate sequence diagrams from Java code. As stated: "if program understanding is the objective, carefully planned tagging is required to produce a highly informative animation" (Lohr and Vratislavsky, 2003).

Another type of tool is presented by Grati et al. (2010). It generates a sequence diagram but from a set of executions traces based on use-case scenarios. Then, those execution traces are automatically aligned to produce a combined trace. A general overview of trace exploration tools is presented by Hamou-Lhadj and Lethbridge (2004). Yet, the presented tools do not use UML sequence diagrams from Java source code, although some use similar notations. Briand et al. (2006) provide a methodology to reverse engineer scenario diagrams — partial sequence diagrams for specific use case scenarios — from dynamic analysis in distributed systems in Java/RMI context. The JavaCallTracer tool (Naqvi, 2011) produces sequence diagrams from Java code. Yet, it offers no integrations with other tools and it is targeted to advanced end users. Trace4J (Inghelbrecht, 2009) seems to have similar objectives to our own, but, when tried, it was not available at the specified address (<http://www.trace4j.com>).

Finally, in (Merdes and Dorsch, 2006) the reader can find an interesting discussion, and also some additional references, related to the development of sequence diagram generators from Java source code.

## 4 Future Work

As future work, the tool will expand to fulfill the missing objectives: (1) integration with other tools for teaching programming at the introductory level; (2) support for a step by step execution of Java code; (3) visualization of each thread role in program execution.

## References

- João Paulo Barros, Luís Biscaia, and Miguel Vitória. Java2Sequence A Tool for the Visualization of Object-Oriented Programs in Introductory Programming. In *Proceedings of the 16th annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11. ACM, 2011. Accepted poster, to appear.
- L.C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *Software Engineering, IEEE Transactions on*, 32(9):642–663, sept. 2006. ISSN 0098-5589.
- H. Grati, H. Sahraoui, and P. Poulin. Extracting Sequence Diagrams from Execution Traces Using Interactive Visualization. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 87–96, oct. 2010.
- Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A Survey of Trace Exploration Tools and Techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, CASCON '04*, pages 42–55. IBM Press, 2004. URL <http://portal.acm.org/citation.cfm?id=1034914.1034918>.
- Cay S. Horstmann and Alexandre de Pellegrin. Violet UML Editor, 2011. URL <http://alexdp.free.fr/violetumleditor/page.php>. Accessed on 2010/04/28.

- Yanic Inghelbrecht. Object-oriented Design with Trace Modeler and Trace4J. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 375–375, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-381-5. URL <http://doi.acm.org/10.1145/1562877.1563017>.
- Michael Kölling. BlueJ – The Interactive Java Environment, 2011. URL <http://www.bluej.org>. Accessed on 2011/04/29.
- Hugo Leroux, Christine Mingins, and Anya Rquil-romanczuk. JACOT: A UML-Based Tool for the Run-Time Inspection of Concurrent Java Programs. In *2nd International Conference on the Principles and Practices of Programming in Java*, 2003.
- K.-P. Lohr and A. Vratislavsky. JAN - Java Animation for Program Understanding. In *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*, pages 67 – 75, oct. 2003.
- G. Malnati, C.M. Cuva, and C. Barberis. JThreadSpy: A Tool for Improving the Effectiveness of Concurrent System Teaching and Learning. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 5, pages 549 –552, dec. 2008.
- Katharina Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 643–646. Springer Berlin / Heidelberg, 2002. URL [http://dx.doi.org/10.1007/3-540-45875-1\\_13](http://dx.doi.org/10.1007/3-540-45875-1_13). 10.1007/3-540-45875-1\_13.
- Matthias Merdes and Dirk Dorsch. Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: a Case Study in Modern Java Development. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, PPPJ '06, pages 125–134, New York, NY, USA, 2006. ACM. ISBN 3-939352-05-5. URL <http://doi.acm.org/10.1145/1168054.1168072>.
- Syed Ali Naqvi. Java Call Trace to UML Sequence Diagram, 2011. URL <http://sourceforge.net/projects/javacalltracer/>. Accessed on 2010/04/28.
- Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, International Seminar*, pages 176–190, London, UK, 2002. Springer-Verlag. ISBN 3-540-43323-6. URL <http://portal.acm.org/citation.cfm?id=647382.724668>.
- OMG. UML 2.x Superstructure Specification, 2011. URL <http://www.omg.org/spec/UML/>. Accessed on 2010/04/28.
- Ervin Ramollari and Dimitris Dranidis. StudentUML: An Educational Tool Supporting Object-Oriented Analysis and Design. In *in Proceedings of the 11th Panhellenic Conference on Informatics*, 2007.
- Diomidis Spinellis. Automated Drawing of UML Diagrams, 2008. URL <http://www.umlgraph.org/>. Accessed on 2010/04/28.
- Scott A. Turner, Manuel A. Pérez-Quiñones, and Stephen H. Edwards. minimUML: A Minimalist Approach to UML Diagramming for Early Computer Science Education. *Journal on Educational Resources in Computing*, 5, December 2005. ISSN 1531-4278. URL <http://doi.acm.org/10.1145/1186639.1186640>.